The Protocol Stack of Application Development

应用软件开发协议栈

谭 喆 · 编著



东南大学出版社
SOUTHEAST UNIVERSITY PRESS

应用软件开发协议栈

谭　喆　编著

东南大学出版社
SOUTHEAST UNIVERSITY PRESS
·南京·

图书在版编目(CIP)数据

应用软件开发协议栈 / 谭喆编著. —南京 : 东南大学出版社,2020. 3

ISBN 978-7-5641-8845-0

Ⅰ. ①应… Ⅱ. ①谭… Ⅲ. ①应用软件—软件开发 Ⅳ. ①TP317

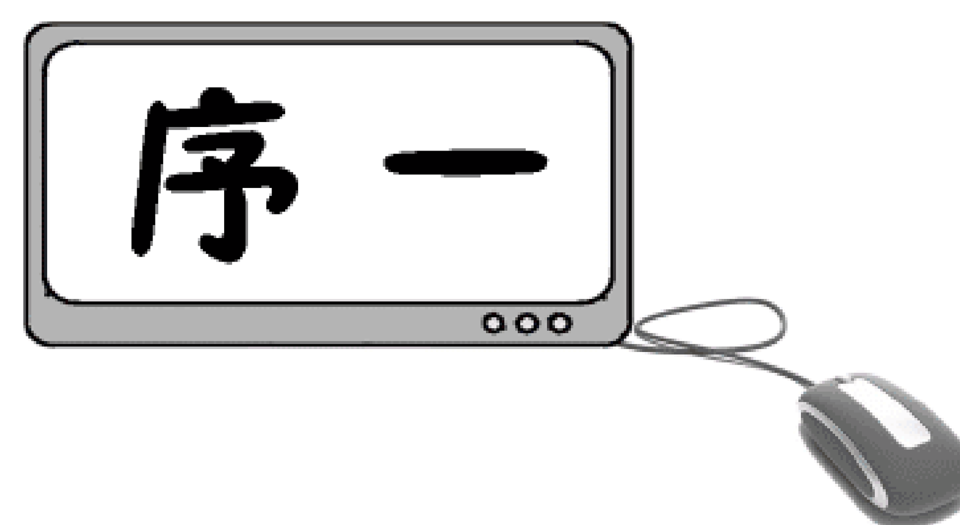
中国版本图书馆 CIP 数据核字(2020)第 031357 号

应用软件开发协议栈

Yingyong Ruanjian Kaifa Xieyizhan

出版发行	东南大学出版社
出 版 人	江建中
社　　址	南京市四牌楼 2 号(邮编 210096)
印　　刷	
经　　销	全国各地新华书店
开　　本	787 mm×1092 mm 1/16
印　　张	39.25
字　　数	857 千字
版 印 次	2020 年 3 月第 1 版 2020 年 3 月第 1 次印刷
书　　号	ISBN 978-7-5641-8845-0
定　　价	99.00 元

*** 本社图书若有印装质量问题,请直接与营销部联系,电话:025—83791830**。



序一

谭喆让我为他工作之余的用心之作《应用软件开发协议栈》作序，我欣然接受邀请，期冀能够鼓励更多优秀的同事勇于钻研实践、乐于分享知识。

近年来，AIoT（智能物联）成了若干企业的追逐目标。但从概念到落地，企业转型为AIoT产品技术型公司需要面对一个又一个困难与挑战。技术能力、行业能力以及人才队伍是实施AIoT战略转型的基础。到底需要什么样的技术能力才能起航AIoT这艘战舰？这是个仁者见仁智者见智的问题。但有一点应该是可以达成共识的：企业需要软件开发领域的全栈能力和拥有这种能力的人才队伍。从这一点来说，《应用软件开发协议栈》一书的出版正逢其时。

智能应用软件是AIoT系统的重要组成部分，其稳定性如何？性能怎么样？是否易于扩展？是否体现了应用使能与数据智能的融合性？诸如这些问题都直接关系到整个AIoT系统的效能发挥，关系到用户的体验，也关系到整个系统在高智能、高并发下的鲁棒性。如何才能具备这种构建与开发一个好的智能应用软件的技术能力？这是AIoT企业必须要回答的问题。

全书从整个信息化系统的“全栈”组成出发，以底层系统的运行机理为主线，以自主可控、安全可用为抓手，详细描述了操作系统、通信协议、安全防护等领域的原理和机制，力求为读者展示软件运行各组成部分的全貌，呈现底层支撑系统和安全防护的知识图谱。根据我的经验，这样详尽的内容，一定会引起开发工程师的共鸣、启发和探讨。

2020年将迎来5G应用元年，AIoT技术将推动人工智能、大数据与物联网在智慧城市和企业数字化转型中的深度应用。力维智联选择AIoT作为企业的核心战略，致力于全智能和高性能的AIoT系统的创新与开发。因此，这本书也源自作者及其团队在AIoT系统开发实践中的技术积累，是一线工作者的经验沉淀与总结。

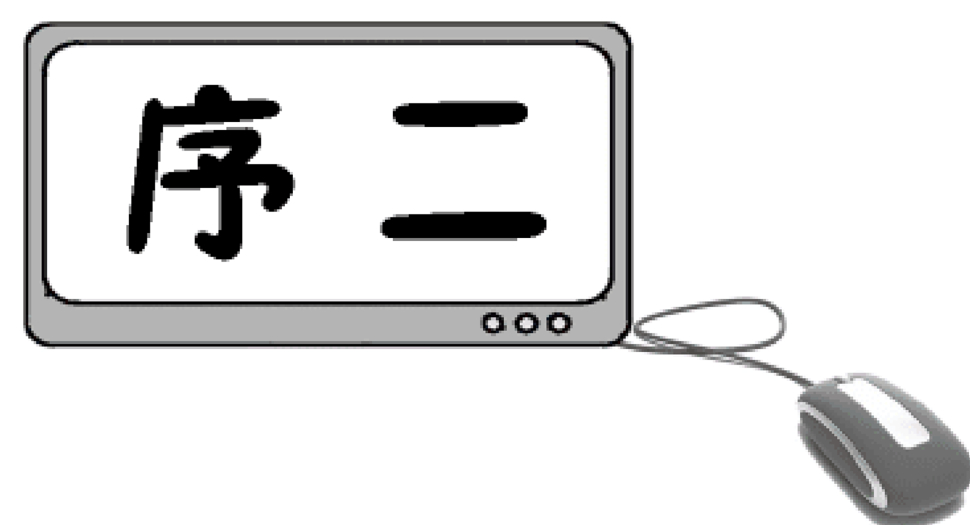
我相信，每一位读者都能够从本书中受益。

徐　明

深圳市人工智能产业协会监事长

深圳力维智联董事长兼 CEO

2020 年 1 月



应用软件开发是一个非常广阔的领域,蕴含了无限的可能。特别是 Intel、AMD、NVIDIA 等传统芯片硬件厂商在做好算力提升的前提下越来越重视软件及其生态,越来越以软件和数据制胜,越来越积极响应眼下时兴的以软硬件解耦为最终目的的 SDN(Software Defined Network,软件定义网络)、SDS(Software Defined Storage,软件定义存储)等技术,凸显了软件,特别是上层应用软件的价值。因此,了解软件的全栈链,理解支撑软件运行的软硬件结构、操作系统、通信协议和防护机制就显得越来越重要。

从另一个角度看,软件是大数据的汇聚方和使用者,任何数据的价值只有通过软件才能实现。因此,怎么保证数据在软件中的合理流向,怎么做好软件中数据资源的安全防护,怎么保证在传输中数据的高效和私密,就成了开发者的 another important subject。

这本书的命名中包含了“协议栈”一词,从计算机专业英语的角度来说,协议就是 protocol,这一般指狭义上的通信协议。但从广义上来说,软件的整个 stack 或者说 chain,它们之间的分层,它们之间的 ABI(Application Binary Interface,应用二进制接口)不也是协议的一种吗?这种协议或者说规约(stipulation),不正是以分层和堆栈的思想支撑着软件的运行吗?因此,这本书以协议栈来命名,就是要描述在这种规约下操作系统是怎么运行的,网络协议有什么特点,系统安全是怎么回事,而这些 subject 在逻辑上又都是以“栈”的形式串联在一起的。

书中把这些相关的技术点总结划分为三个区间(interval),这一点我也是经过深思熟虑并与作者进行过探讨的。信息技术是一片浩瀚的大海,如果把这片大海映射到一个坐标轴上,可以看作正负无穷的无限区间。在这片大区间中,我们仅仅选取了比较有心得,同时也与我们从事的领域联系比较紧密的几小块来描述。这几个区间是有 overlap 的,但更多的是不相交的 interval,同时也是 finite interval。这几个区间分别描述了:

- 操作系统的 Kernel 机制,解决怎样支撑软件运行的问题。
- 系统的设备驱动机制,描述了操作系统和外围设备的联系与互动,这也同样支撑了软件的运行。
- 通信协议和网络安全,这是与开发人员联系最为紧密的部分。

我曾经提出过软件开发的“八可”:可融合、可对接、可扩展、可分层、可堆叠、可调试、可发布和可监控,实践证明这是符合技术趋势的。而实现这一切的基础和保证,就是我们的软件协议栈。

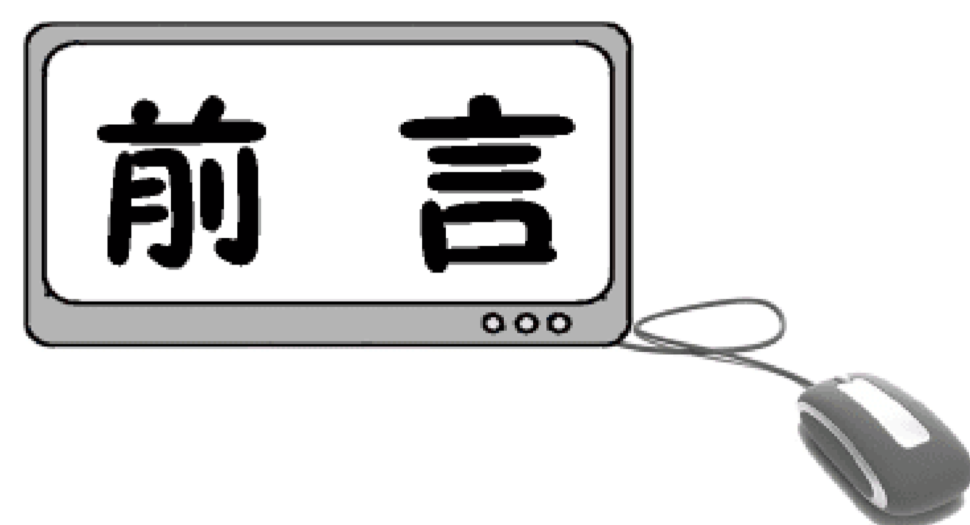
作者担任软件开发的架构师和产品总工多年,曾经参与过数十个大规模视联网软件的总体设计和系统开发,同时积极探索和学习相关领域的技术,积累了许多切身经验和体会,也非常注重分享与传承,发表过许多论文和专利。今天,他将这些经验和体会作为积累的一部分系统地总结出来介绍给大家,供大家参考。这也是我第一次看到在一本书中如此明确地提出软件生态、平台算力、软硬件解耦和软件全栈链的思想。希望本书能为软件开发者提供一定的参考和启发。

曹友盛

美国超算中心超级计算科学博士后

深圳力维智联 CTO

2020 年 1 月



前言

应用软件技术发展到今天,从功能的角度来说越来越专业化,领域越来越细分;从架构上来说越来越分层化和解耦化。特别是近些年来,随着大数据、云计算、微服务等技术的发展,应用软件越来越高层化,也就是说在整个软件堆栈中居于越来越高层的位置。这是由以下几方面的原因造成的:

首先,计算平台越来越通用化,开发者越来越不需要面向不同的处理器架构进行编程。

其次,操作系统、虚拟机软件、容器软件等软件平台愈发成熟、愈发全能化、愈发跨平台化,使开发者越来越不需要考虑操作系统的实现机制与适配问题。

再次,诸多开源软件对于通信机制、线程管理、内存安全、I/O 读写等基础功能均有良好的封装,使开发者在面对这些功能时无需再考虑复杂的同步管理机制。

最后,应用软件的许多基础设施越来越自动化,越来越完善,使应用软件无需应对过多业务诉求以外的羁绊。

以上这几方面的原因既是软件发展的必然之势,也是技术、架构、业务、领域等多方面解耦和深化的大势所趋,为开发者带来了实实在在的实惠和便利,并成为细分领域应用软件专业化使能的源动力。

从软件堆栈的角度来看,应用软件的层次和位置越来越高,并且也迎来了极大的横向扩展。如果把软件体系比喻为一棵树,应用软件就像开枝散叶的枝干和树叶,越来越茂盛,越来越细分;而位居应用软件之下的通信框架、基础设施、操作系统甚至计算平台却越来越归一化,就像大树的树干越来越粗,却没有“旁逸斜出”。

这同时也带来了一个问题,应用软件的开发者越来越关注业务和架构,越来越多地使用成熟的框架,而对这些框架是怎么实现的、设备是怎么接入和抽象的、资源是怎么调度和分配的、人机交互是怎么实现的、应用软件的生命周期是怎么管理的这些底层和深层的问题茫然无所知,既不感兴趣也不是很关注,更有一种“八竿子打不着”的距离感,因为关注于业务和领域的应用软件在堆栈中离这些底层机制和框架越来越远了。

但是,作为应用软件开发者,对这些成熟的架构和机制,我们不能仅仅拘泥于“会用”和“好用”,满足于“用会”和“用好”,更要着力于“用懂”和“用深”。

我们研究一个软件,既要研究其上层实现机制,亦要研究其底层技术原理,同时考虑到与外部交互,还要研究通信相关的技术,因而不外乎自底向上的这几个方面:操作系统、通信技术、通信协议、网络安全、上层业务。这就需要站在软件堆栈的角度把事情讲得明白、讲得贯通、讲得全面,还要讲得有一定的深度。这在表面上看是“降层”,但实际上是“升维”,任何高层的原理必须依赖于底层的机制,只有深刻领悟了底层的机制,才能更好地“玩转”高层。底层的往往也是基础的,基础的研究可能在短期内看不到成效,但必然会在潜移默化中为我们带来意想不到的收益。而从技术深耕的角度来说,我们也应该有这个理想去下钻更深层的原理。

本书按照自底向上的顺序和思路,对软件堆栈中的各层做了梳理,并借助数学中实数区间(real number interval)的概念将这些技术分为三个有限区间(finite interval),对于每个区间,都有一定的主线和脉络贯穿其中。区间之间有一定的重合(overlap),而在区间之外则是更广阔的无限空间(infinite interval),代表了信息技术的无穷延展。

- 第一区间:描述操作系统核心机制,并以线程堆栈的动态平衡为主线来分析这些核心机制。
- 第二区间:讲述操作系统外围机制,特别是操作系统的驱动框架,并以此为基础分析了网络通信、文件系统等 I/O 流程和操作系统安全特性。
- 第三区间:着眼于通信,这是与应用软件联系最为紧密的部分,可以总结为四个关键词,即通信模型、通信技术、通信协议、通信安全。

这里,我们也提出“全栈”的思想。但这里的全栈不是什么开发语言从前端到后端的“全栈”,而是从底层平台到上层应用的软件堆栈的“全栈”。这种“全栈”可能不会立即为我们带来直接的收益,但却能为我们带来融会贯通的酣畅感和了然于胸的满足感。

具体到软件技术这个我们熟悉的领域,它的一日千里、它的枝繁叶茂、它的步步高升使人炫目,也让我们有了一种不得不重新审视的陌生感和敬畏感。但越是陌生,我们越要刨根寻底,于千变万化中找寻一定之规;越是敬畏,我们越要追踪溯源,于枝繁叶茂中找寻不变之基。

本书命名为《应用软件开发协议栈》,但这里我们既不是讲述如何开发软件也不是仅仅描述通信协议的格式,既不是讲语法也不是讲操作,而是讲内核机理,讲运行过程,这些原理和框架可以被称作“软件开发协议”,而“栈”也契合了软件开发分层化的“堆叠”思想。我们认为,贯通软件堆栈中各层的联系才是打通软件开发的“任督二脉”,理解每一层框架的运行机制才是软件开发的“化骨绵掌”。因此读者也不必过于纠结文中诸如函数名这样的过于细节的内容。

下钻也许是艰难痛苦的,但正如人们常说的,做难事必有所得。通晓了底层的原理,必然使我们具备高屋建瓴的宏观视野,而这需要坚持不懈的努力和孜孜以求的定力。

希望本书能为应用软件的开发,特别是安防和物联网领域的软件开发做出有益的贡献。

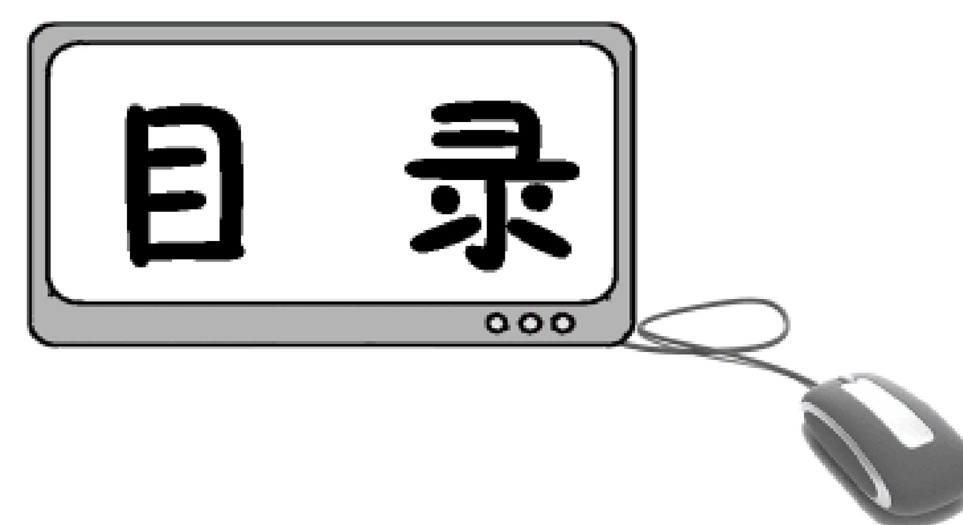
即使是产品经理,如果能从底层原理来看待应用软件的开发则必然会有不一样的发现。

这种以协议栈为观察视角、以框架和技术为思辨逻辑的方法论也是调和产品经理与程序员之间“矛盾”、加强彼此“心灵沟通”的不二法门。

感谢力维智联徐明董事长、曹友盛博士等领导对我的极大支持和鼓励，他们对行业的理解、对技术的精研和对趋势的把握深深影响了我，并成为我矢志不渝的动力源泉。

最后，感谢我的家人，特别是我的女儿，在我奋笔疾书的时候没有怎么打扰我，顶多就是让我陪她搭乐高，真是个乖巧的小朋友。

2019 年 10 月 31 日于南京



第一区间　Windows 系统内核机制

第二区间　Windows 驱动体系

第三区间 应用软件通信机制

第一区间

Windows系统内核机制

第 1 章　CPU 体系结构

CPU(Central Processing Unit),即中央处理器,是计算机最核心的部件。应用软件需要操作系统来承载,而操作系统需要 CPU、主存、磁盘等硬件来承载和运行,CPU 是"盘活"整个硬件系统的大脑。

本章将按照图 1-1 所示的提纲介绍 CPU 的指令集结构、体系结构、存储结构等内容。

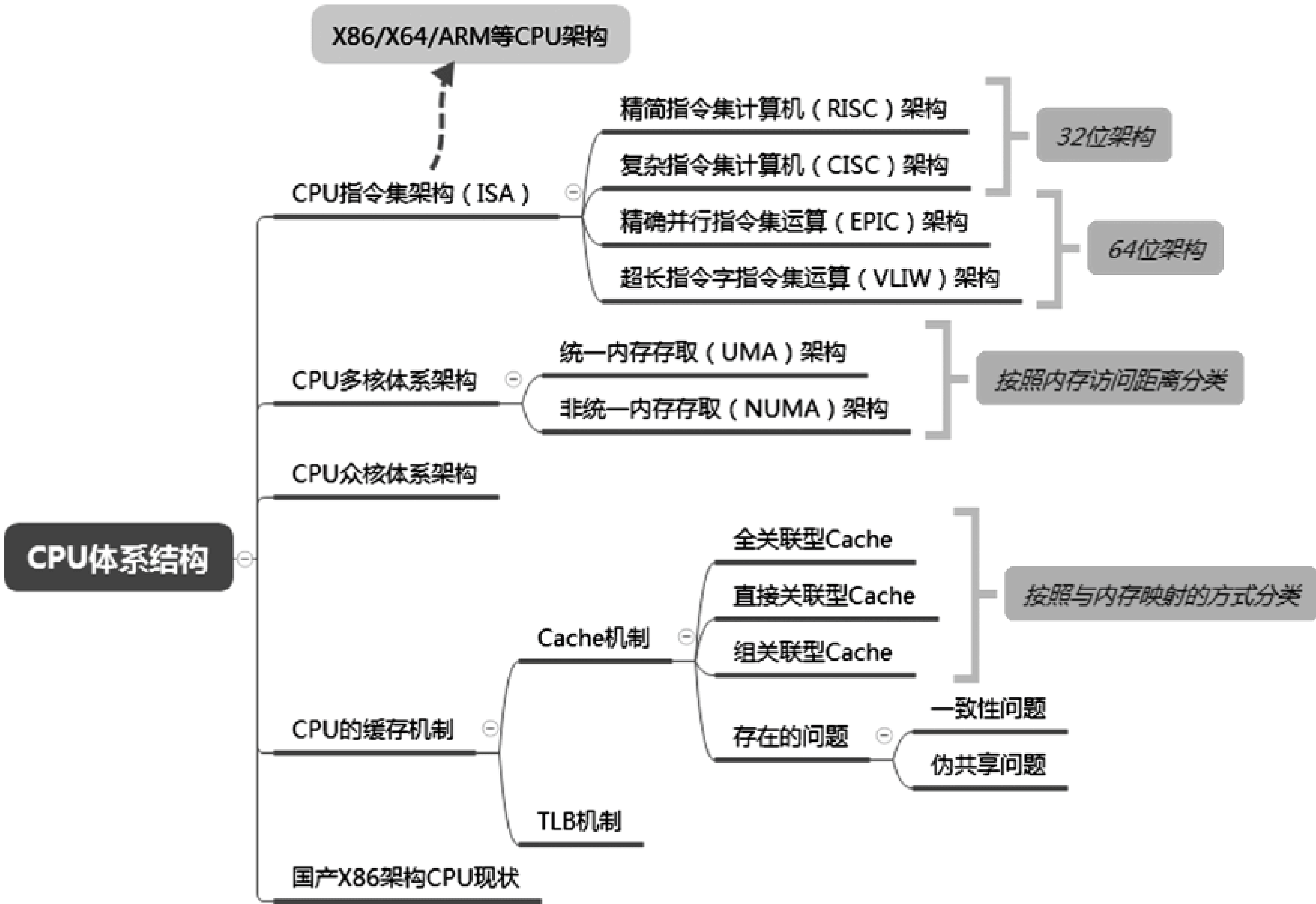


图 1-1　本章提纲

在讲述 CPU 体系结构之前,我们先来看看 CPU 的相关技术和概念。

1.1　相关技术和概念

微码:microcode,一般 CPU 对于指令的操作可分解为取指、解码和执行。在解码阶段,对于长短不一致的复杂指令集(CISC)一般分解为若干条长度一致的更精简的指令,以便于加快解码速度和提高并行性,这些指令被称为微码。通常我们所说的对 CPU"打补丁"大多也是对微码"打补丁"。微码在计算机语言堆栈中处于最下层(汇编指令的下一层),如图 1-2 所示。

多核技术:CMP(Chip Multi Processors)技术,也称为单芯片多处理器/片上多处理器技术,即在同一个芯片上集成多个核心(Core),这些核心单元负责运算、接收和存储命令、处理数据等工作。核心单元除了自己固有的逻辑结构外,还包括执行单元 EU、一级 Cache、二级

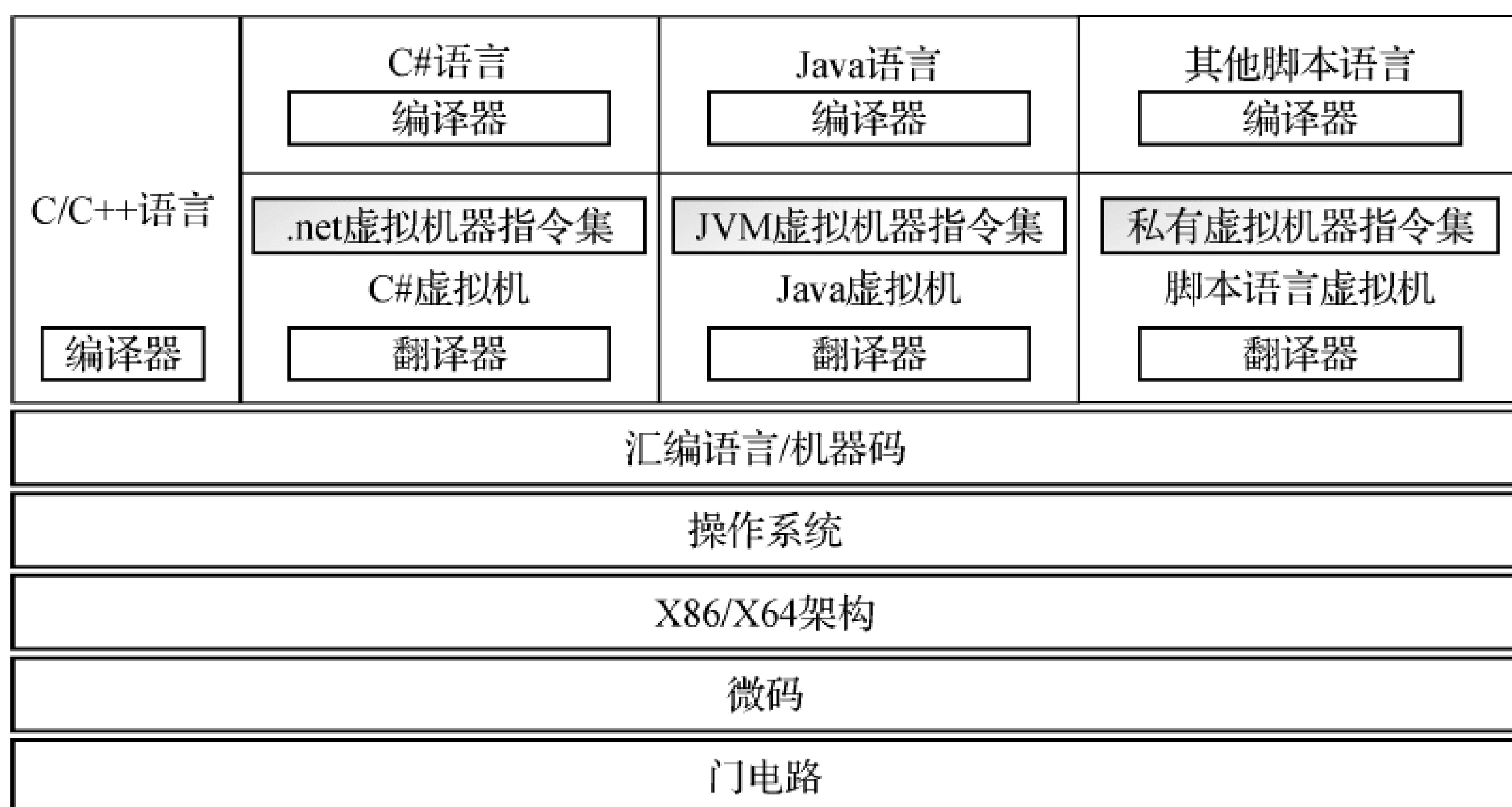


图 1－2　计算机语言堆栈结构

Cache、总线接口等。

众核技术：拥有多于 8 个核心的 CPU 称为众核。

多路技术：在同一个主板上集成多个 CPU 处理器的技术。一般家用台式计算机/笔记本计算机都采用多核单 CPU，多路技术常见于服务器/小型机系统。这里要注意区分多核心与多 CPU 的概念，一个 CPU 可以包含多个核心，这被称为多核。

向量化技术：使用同一条指令同时操作多个数据的技术。当然，使用向量化技术需要应用程序采用并行化代码编写。

SIMD：Single Instruction Multiple Data，即单指令多数据流。这是一种对一组数据（数据向量）中的每一个分别执行相同的操作从而实现空间并行性的技术，常见于 GPU（Graphics Processing Unit，图形处理单元）计算中，如图 1－3 所示。SIMD 要求数据统一输入输出，其指令操作寄存器相对于通用寄存器而言宽度更宽。

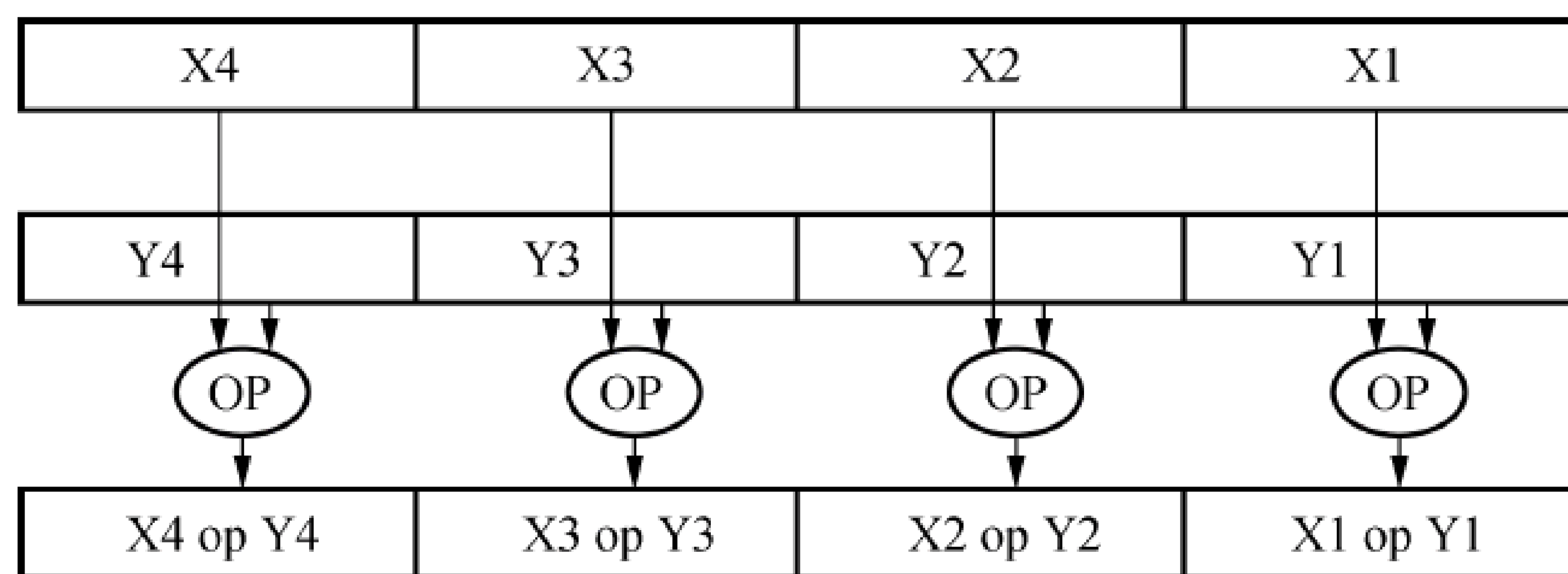


图 1－3　SIMD 操作

SMP：Symmetric Multi Processor，即对称多处理器。这种架构下各 CPU 之间共享内存子系统以及总线结构，每一个点访问内存的速度和距离是一样的，CPU 地位对等，“对称”之名由此而得。SMP 架构也被称为统一内存存取（Uniform Memory Access，UMA）架构。采用 SMP 架构的 CPU 核心之间的缓存数据同步是由操作系统完成的。

NUMA：None Uniform Memory Access，即非统一内存存取。这种架构下 CPU 有远近之





分，访问内存的速度和距离不一致。

MPP：Massive Parallel Processing，即海量并行处理。这是一种系统外扩展（非 CPU 中扩展）的技术手段，即多个 SMP 服务器通过一定的节点互联网络进行连接，分布式协同工作，完成相同的任务，但从用户的角度来看这是一个服务器系统，拥有统一的计算入口。

AMP/异构计算：Asymmetric Multi Processor，即非对称多处理器，例如超算。在这种架构下采用 GPU 或者 FPGA（Field Programmable Gate Array，现场可编程门阵列）作为协处理器，CPU 作为通用计算单元并总体协调各异构处理器，负责向这些协处理器“喂”数据或者协调它们统一工作，CPU 既是协处理器的供给侧，也是协处理器的消费侧。

超线程技术：Hyper-Threading Technology，是 Intel 为了更好地利用 CPU 资源而提出的一项技术。超线程技术的学名叫同步多线程（Simulate Multi Threading，SMT）技术，即在每个 CPU 的物理核上虚拟出多个逻辑核，每个逻辑核有自己的 CPU 状态，但是运算单元和片上系统（System-on-a-Chip）缓存是共享的。在 CPU 中，运算处理单元（Processing Unit，PU）负责诸如加减乘除这样的运算执行功能；架构状态（Architectural State，AS）单元负责执行逻辑和调度方面的操作，比如控制内存访问等。超线程技术就是通过 AS 单元模拟出逻辑核的，是对 CPU 核的虚拟化。

假如一个 CPU 有 4 个物理 Core，每个 Core 有两个 HT（Hyper Thread），从操作系统角度来看，每个超线程都是一个逻辑核，因此一共有 8 个逻辑 CPU。图 1－4 所示的就是单个双核处理器使用超线程技术后的情况（双核心四线程）。一个物理 Core 中一般有两个 AS 和一个 PU，这两个 AS 共享一个 PU。AS 与 PU 的关系就像饭店服务员与大厨，就是靠这种“多个服务员围着一个大厨转”的机制，使大厨“满负荷运行”。

乱序执行技术：Out of Order Execution，即 CPU 不按照指令的顺序执行，而是在执行当前指令时将可以提前执行的指令也放入执行单元执行，以加快指令运行速度。

三级 Cache（缓存）：CPU 采用三级缓存机制来平衡计算速度（CPU）与存储速度（内存）的巨大差距。Cache 的读写速度远快于内存，但是要比寄存器存取慢。其中第一级 Cache 又分为指令缓存（I-Cache）和数据缓存（D-Cache），第二级 Cache 不分指令缓存和数据缓存，第三级缓存则为所有 CPU 共享。上述三级缓存与 CPU 逻辑核心的关系如图 1－4 所示。

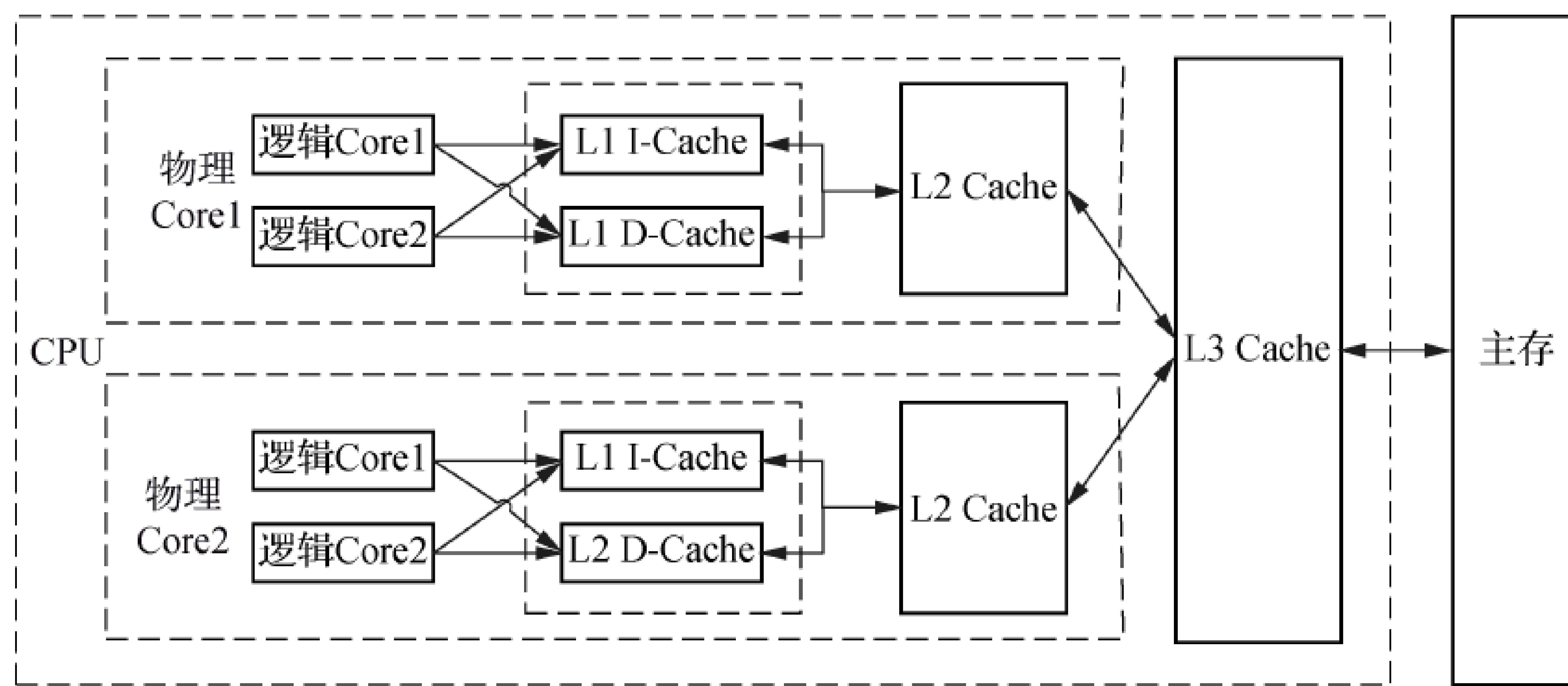
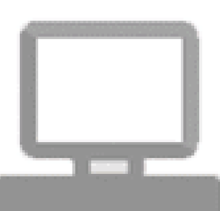


图 1－4　一个双物理核双超线程 CPU 的三级缓存示意图





之所以要分成 I-Cache 和 D-Cache，是因为 I-Cache 大多都是顺序取指，而 D-Cache 的访问模式变化却比较大。此外 D-Cache 包括了读写两方面的操作，而 I-Cache 则只有读操作，因此将这两类缓存分开以避免互相干扰。

RAM：Random Access Memory，即随机访问存储器，多用于主存。

SRAM：Static Random Access Memory，即静态随机访问存储器，存取速度非常快，通常用于处理器内部的 Cache，其内部有一块芯片以维持信息。

DRAM：Dynamic Random Access Memory，即动态随机访问存储器，目前该技术已被淘汰。

SDRAM：Synchronous Dynamic Random Access Memory，同步动态随机访问存储器，用于主存，基于同步时钟进行同步，采用 SDRAM 结构的系统处理器和内存共享一个时钟周期，工作速度同步。目前该技术逐渐被 DDR 技术取代。

DDR：双数据速率 SDRAM（Double Data Rate SDRAM），是 SDRAM 的升级版本，用于主存，目前已经发展到 DDR4（第四代 DDR）。

总线：总线用于内存、I/O 设备与 CPU 之间的数据传输，内存与 I/O 设备共享这条信道。旧一点的 CPU 采用前端总线（Front Side Bus，FSB）技术，但是新一代的 Intel 处理器已经内置了内存控制器，因此需要把前端总线的功能一分为二，一条分通道连接 CPU 与内存，另一条分通道连接 CPU 与 I/O 设备，前者称为内存总线，后者称为快速通道互联（Quick Path Interconnect，QPI）。

CPU 亲和性：CPU Affinity，这是一种调度属性，它可以将一个线程"绑定"到一个或一组 CPU 上。CPU 与线程绑定有以下好处：

- 可以提高 Cache 的命中率以减少内存访问损耗；
- 可以减少线程切换带来的上下文切换开销（单方面绑定线程与 CPU 还不够，还应将 CPU 从其他 CPU 的调度队列中移除）；
- 可以提高线程运行的实时性，使之一直运行而不被调度到线程就绪队列。

流水线技术：流水线通常由取指、译码、执行以及 Load/Store 等单元组成，流水线技术是一种将每条指令分解为多步（取指、译码等）并使各步操作重叠，从而实现几条指令并行处理的技术，如图 1－5 所示。程序中的指令仍是一条条地顺序执行，但可以预先取若干条指令，并在当前指令尚未执行完时，提前启动后续指令的另一些操作步骤。流水线技术一般是通过增加计算机硬件来实现的，例如预取指令的电路等。

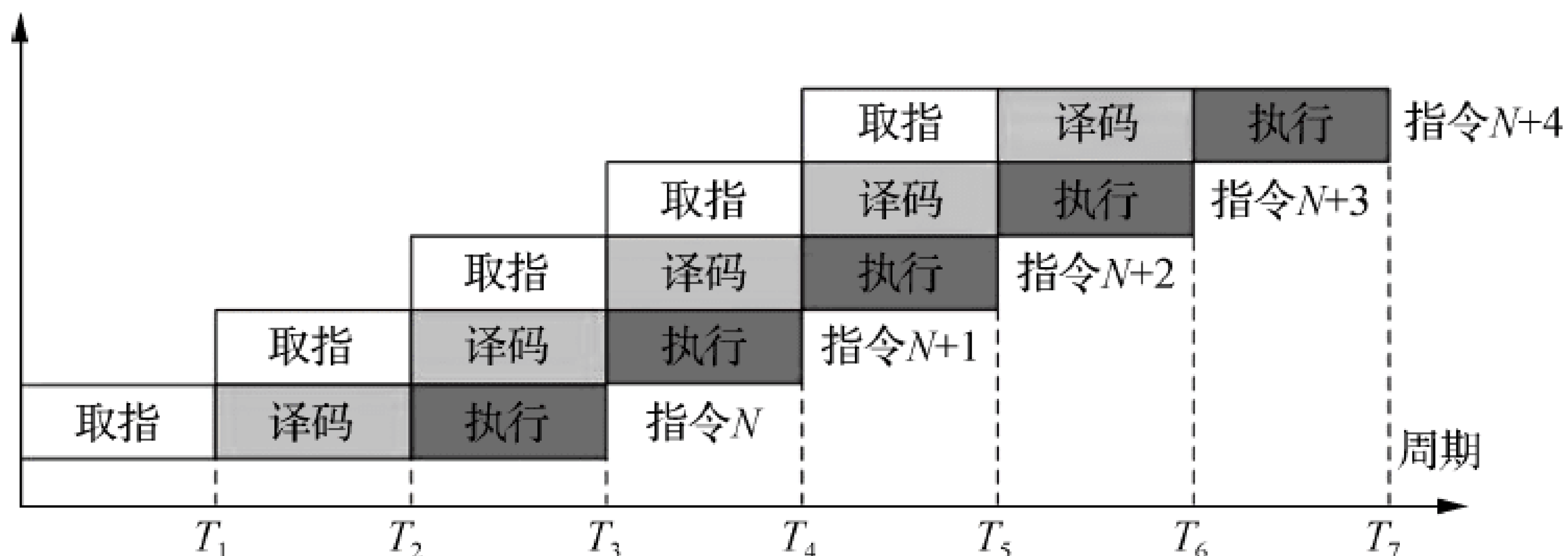


图 1－5　流水线执行操作

超级流水线技术:超级流水线以增加流水线级数的方法来缩短执行周期,相同的时间内超级流水线执行了更多的机器指令。超级流水线配置了多个功能部件和指令译码电路,采用多条流水线并行处理,并具有多个寄存器端口和总线,可以同时执行多个操作,因此比普通流水线执行得更快,在一个机器周期内可以流出多条指令。

超级流水线技术是通过细化流水、提高主频,使得在一个机器周期内完成一个甚至多个操作实现的,其实质是以时间换取空间。

超标量技术:超标量(superscalar)技术是指在 CPU 中有一条以上的流水线,每个时钟周期内可以完成一条以上指令的技术,如图 1-6 所示。超标量技术的实质是以空间换取时间。

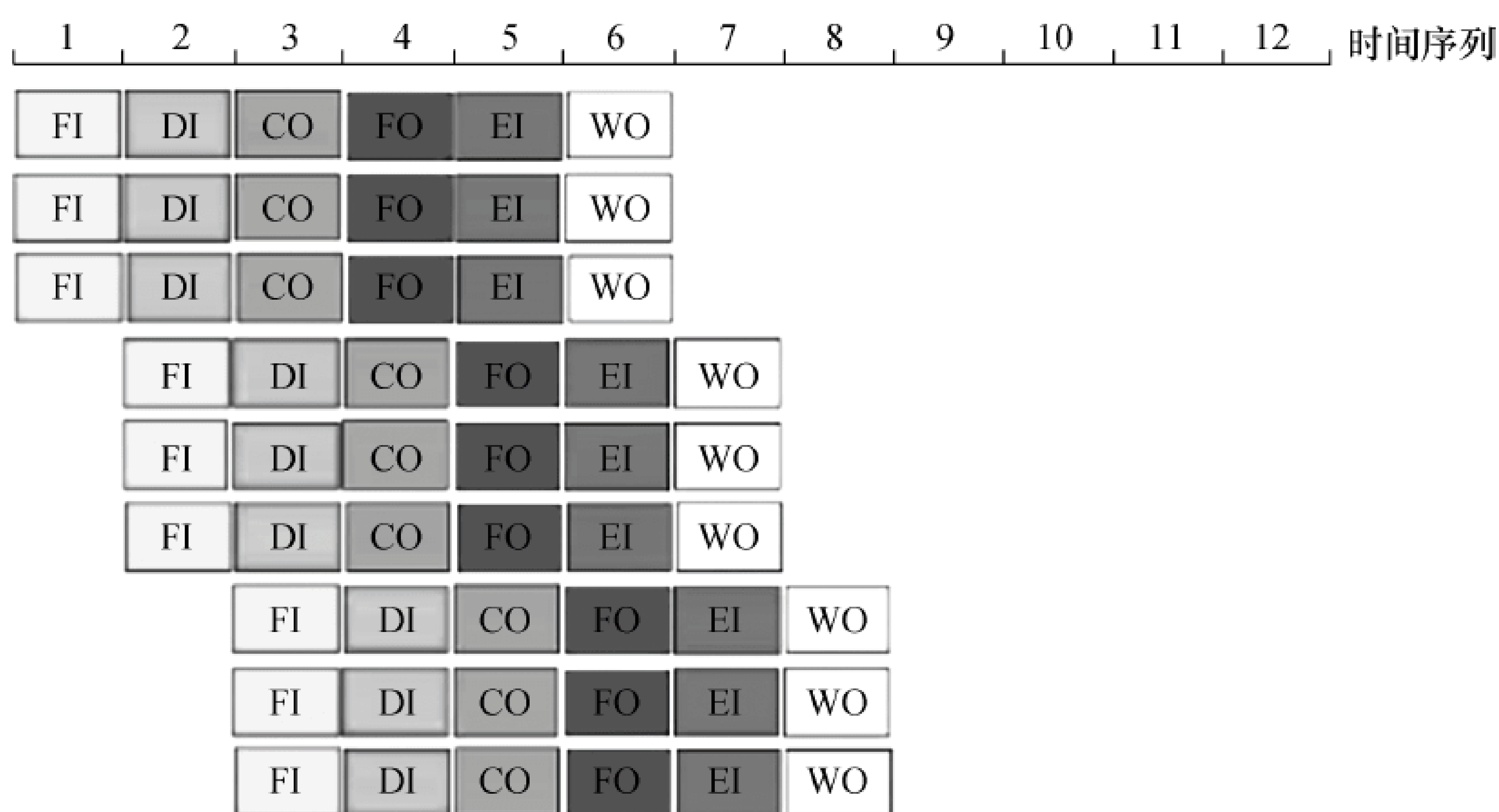


图 1-6　超标量执行操作

1.2　微处理器指令集架构

精简指令集计算机(Reduced Instruction Set Computer, RISC)和复杂指令集计算机(Complex Instruction Set Computer, CISC)是 CPU 最常见的设计模式,而 CPU 的设计模式也被称作微处理器的指令集架构(Instruction Set Architecture, ISA)。简单地理解,ISA 就是一个抽象的机器架构,这个抽象的架构解耦了编程所需要的软件和硬件体系,如此一来,CPU 也就演变成了由指令集、微结构和物理设备组成的三层结构。除了 RISC 和 CISC, ISA 还包括精确并行指令集运算(Explicitly Parallel Instruction Code, EPIC)和超长指令字指令集运算(Very Long Instruction Word, VLIW)两个种类,它们主要针对 64 位处理器。

顾名思义,所谓精简指令集,其指令数目和寻址方式都做了精简,使其实现更容易,所有指令的格式都是一致的,所有指令的指令周期也相同,指令并行执行程度更好,编译器的效率更高,因而更适合诸如 I/O 并发量大、需要并行处理的应用场景;而复杂指令集是为了方便软件编程和提高程序运行速度而采取的不断增加实现了复杂功能的指令和多种灵活的编

址方式的设计模式，甚至某些指令可支持高级语言语句归类后的复杂操作，比较适合通用计算的应用场景。

精简指令集执行的都是等长的指令，因此执行效率较高，性能也比较稳定，基于 RISC 的 CPU 工艺也相对简单；而复杂指令集的指令不等长，执行时需要进行识别和分割，因此执行效率没有 RISC 高，且基于 CISC 的 CPU 工艺较为复杂，部件比较多。两种指令集的区别如表 1-1 所示。

表 1-1　复杂指令集与精简指令集的比较

比较内容	CISC	RISC
指令系统	复杂、庞大	简单、精简
指令数目	一般大于 200	一般小于 100
指令格式	一般大于 4	一般小于 4
寻址方式	一般大于 4	一般小于 4
指令字长	不固定	等长
可访存指令	不加限制	只有取数/存数指令
各种指令使用频率	差异较大	差异较小
各种指令执行时间	相差很大	绝大多数在一个周期内完成
优化编译实现	很难	较容易
控制器实现方式	绝大多数为微程序控制	绝大多数为硬布线控制
软件系统开发时间	较短	较长

Windows 目前支持 CISC 体系结构的 CPU，包括 X86 和 X64 这两种类型。其中，X86（IA-32，Intel Architecture 32，也被称为 X86-32）架构是 Intel 和 AMD 的 32 位 CPU 的统一架构。说是架构，其实是一套指令集和配套的寄存器结构，也就是说 CPU 对这个体系架构的指令集是“可认知”的。

X64（X86-64）的另一个叫法是 AMD64，这是一种采用 VLIW 指令集设计的处理器体系。X86 和 X64 的主要区别还是操作系统位数的区别（32 位还是 64 位）。沿着这种思路 CPU 的设计就有两条路线可以选择：要么设计一套完全兼容 X86 体系的 CPU 架构，要么设计一套完全不兼容 X86 的新的体系架构。AMD 首先设计出了兼容 X86 的体系结构，并命名为 AMD64，而后 Intel 也设计出了不兼容 X86 的全新体系结构，命名为 IA-64，其采用的是 EPIC 指令集架构。看起来当然还是兼容 X86 的体系结构比较受欢迎。可能后来 Intel 也回过味儿来了，也开始支持 AMD64 体系结构，并命名为 X64，这就是 X64 的由来。X64 不但扩展了新的指令集，寄存器数量也大大增加，并且通用寄存器的位数从 32 位扩展到了 64 位（Intel 和 AMD 之间存在交叉专利授权，你可以用我的专利技术，我也可以用你的专利技术，大家重要程度等同，离了谁也活不了，因此 Intel 可以使用 AMD64 架构，但是 X64 与 AMD64 在指令方面有细微的差别，这里暂时忽略）。

而IA-64(安腾处理器架构)其实是Intel与惠普联合开发的架构,采用的是EPIC指令集架构,这种架构脱胎于VLIW,更偏向于RISC体系结构,并且只适用于Intel的安腾(Itanium)系列处理器,与X86指令集不兼容。安腾处理器是较为高端的处理器,在普通的商用服务器市场并不多见。IA-64架构提供了128个整数寄存器、128个浮点寄存器、64个单比特预测器和8个分支寄存器,其中浮点寄存器长度为82bit,因此也具有较强的浮点运算性能。可以看出,IA-64架构仅仅在处理器上就比X64要多出许多,其性能和定位也都因此更加高端。

其实无论是Intel还是AMD,现代CPU本质上都是精简指令集架构的。原因就是复杂指令集的处理器在其核心外围电路中会将复杂指令翻译成精简指令,然后送到精简指令核心中处理。当然,习惯上我们还是把X86架构的处理器看作CISC架构的。那么什么样的CPU是直接采用精简指令集架构呢?我们看图1-7。



图1-7 RISC架构处理器分类

精简指令集一般具有以下特点:

- 流水线及常用指令可以用硬件执行;
- 大部分指令利用寄存器缓存,速度快;
- 存/取数据的指令分开执行,处理器可以完成更多工作。

由图1-7可知,类似PowerPC、MIPS(Microprocessor without Interlocked Piped Stages,无内部互锁流水级的微处理器)、ARM(Advanced RISC Machines,高级精简指令集机器)等架构的处理器是采用精简指令集的。而这些处理器架构也多用于工控设备、物联网设备以及移动端,例如ARM架构更适合低功耗设备的计算任务,且为了进一步降低功耗,ARM采用了大小核兼容的设计思路,大核心高性能高功耗,小核心低性能低功耗。另外,ARM架构比较单纯,更容易与异构处理器(GPU、FPGA等)通过SoC的方式整合在一起。

由于ARM架构处理器节能的特点,它非常适用于移动通信领域。飞腾系列处理器FT2000/FT2000+也是基于ARM架构的,虽然在单核性能上与Intel相比还有不小差距,但在多核性能上已经逐渐追平Intel同级产品,在军事和超算领域有着广泛应用,而且也是国内首款自主可控的CPU。PowerPC处理器同样具有优异的性能和较低的能耗、散热量,其多用于嵌入式设备的中央处理器。

ARM采用知识产权(Intellectual Property,IP)授权的模式,收取一次性技术授权费用和版税。总体来讲有三种具体方式:处理器授权、POP(Processor Optimization Pack,处理器优化

包）授权和架构授权。处理器授权方式是指授权合作厂商使用 ARM 设计好的处理器，对方不能改变原有设计，但可以根据自己的需要调整产品的频率、功耗等。POP 授权是指 ARM 出售优化后的处理器给授权合作厂商，方便其在特定工艺下设计、生产出性能有保证的处理器，这是一种更高级的授权方式。架构授权是指 ARM 会授权合作厂商使用自己的架构，方便其根据自己的需要来设计处理器，这是最高级的授权方式。目前 ARM 对高通、苹果、华为和 NVIDIA 开放了架构授权。

不过除了 ARM 之外，其他的架构都不能直接支持 Windows 操作系统的高阶版本，而且包括诸如编译器、调试器、应用软件等在内的生态系统远不如 X86 和 ARM 健壮，我们在这里不进行过多描述。

1.3　UMA 与 NUMA 架构

在多路技术体系中，按照内存的存取距离来分存在 UMA（SMP）和 NUMA 两种架构。

UMA 架构，即统一内存存取架构（Uniform Memory Access Architecture），在一些资料中也称为 SMP（对称多处理器）架构。在这种架构下，各 CPU 地位均等，访问内存的速度也一致，通过一条系统总线访问主存等资源。这种架构下 CPU 的数量是有限的，因为共享的总线频率是有限的，带宽是有限的，CPU 多，内存等资源的访问就会冲突，反而浪费了 CPU 资源。实验表明，UMA 架构配置 2 ~ 4 个 CPU 时资源利用率最高。UMA 架构如图 1 – 8 所示。

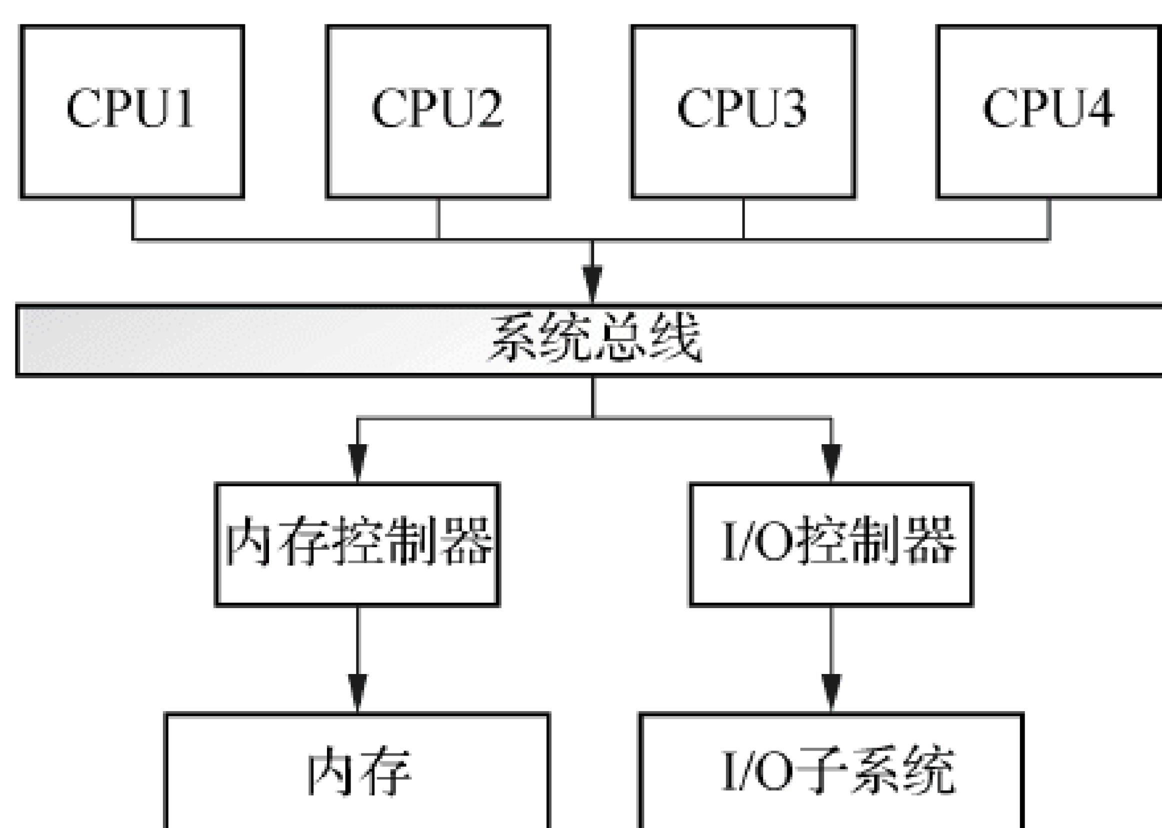


图 1 – 8　UMA 架构示意图

NUMA 架构，即非统一内存存取架构（Non Uniform Memory Access Architecture），起源于 AMD 的 Opteron 架构，它是为了解决 UMA 的扩展性问题而提出的一种架构。在这种架构下，CPU 访问内存有远近之分，因此访问速度是不一样的。但是 NUMA 架构的扩展性很好，上百个 CPU 可以在 NUMA 架构下组合在一起，中间通过高速交换矩阵通信。NUMA 架构如图 1 – 9 所示。

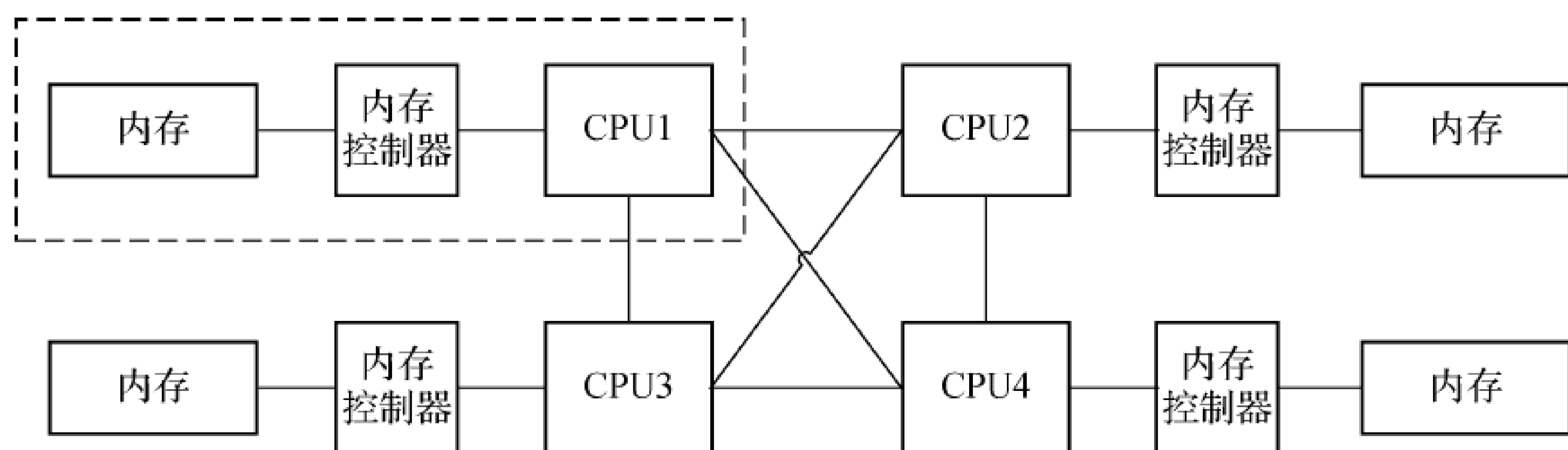


图 1 – 9　NUMA 架构示意图

在图 1－9 中每个 CPU 的内存合起来构成了物理内存空间的全部，且每块内存对于 CPU 都是可以直接寻址访问的。CPU 访问本地内存（虚线框中所示）是非常快的，但因为各 CPU 之间要通过系统总线或者其他高速交换结构通信，因此访问远端的内存就没有那么快了。在进行软件开发的时候，也应尽量避免访问远端内存，例如可以通过指定 CPU 亲和性的方式绑定线程与 CPU。NUMA 虽然一定程度上解决了 UMA 架构的扩展性问题，但是其性能却不是随着 CPU 数量的增加而线性提高的，这主要还是因为对于远端内存的访问开销较大。

无论是 UMA 还是 NUMA 架构，都需要操作系统对此进行支持，包括处理器间通信、处理器核心之间通信和同步等。另外，总线技术也伴随着 CPU 的发展而发展。例如 Intel 提出的 QPI（快速通道互联）总线较好地解决了多处理器之间互联的问题。QPI 技术摒弃了通过前端总线连接到北桥的思想，转而通过基于包传输的串行式高速连接协议直接进行 CPU 间的点对点连接，具有很高的交换速度。

我们有时也把 MPP（海量并行处理）看作多路体系下的一种技术架构。MPP 架构各个节点之间通过网络进行互联，每个节点的处理器一般是 UMA 架构，且本地节点的 CPU 不能直接访问远端节点的内存，它们之间的数据共享要通过上层软件实现。MPP 架构更为宏观，与 UMA/NUMA 在一块板子上进行 CPU 扩展的方式不同，MPP 着眼于服务器的并联扩展，通过软件机制平衡计算负载。

1.4　众核体系架构

无论是 SMP/UMA 还是 NUMA，都是多路技术条件下的架构。前文说过，所谓多路是指一个主板上集成了多个 CPU，因此才会存在 CPU 访问内存的不对等性。多路技术一般用在服务器或者小型机系统中，如果操作系统调度不好，很可能会出现图 1－10 那样的情况（多人围观一人干活）。

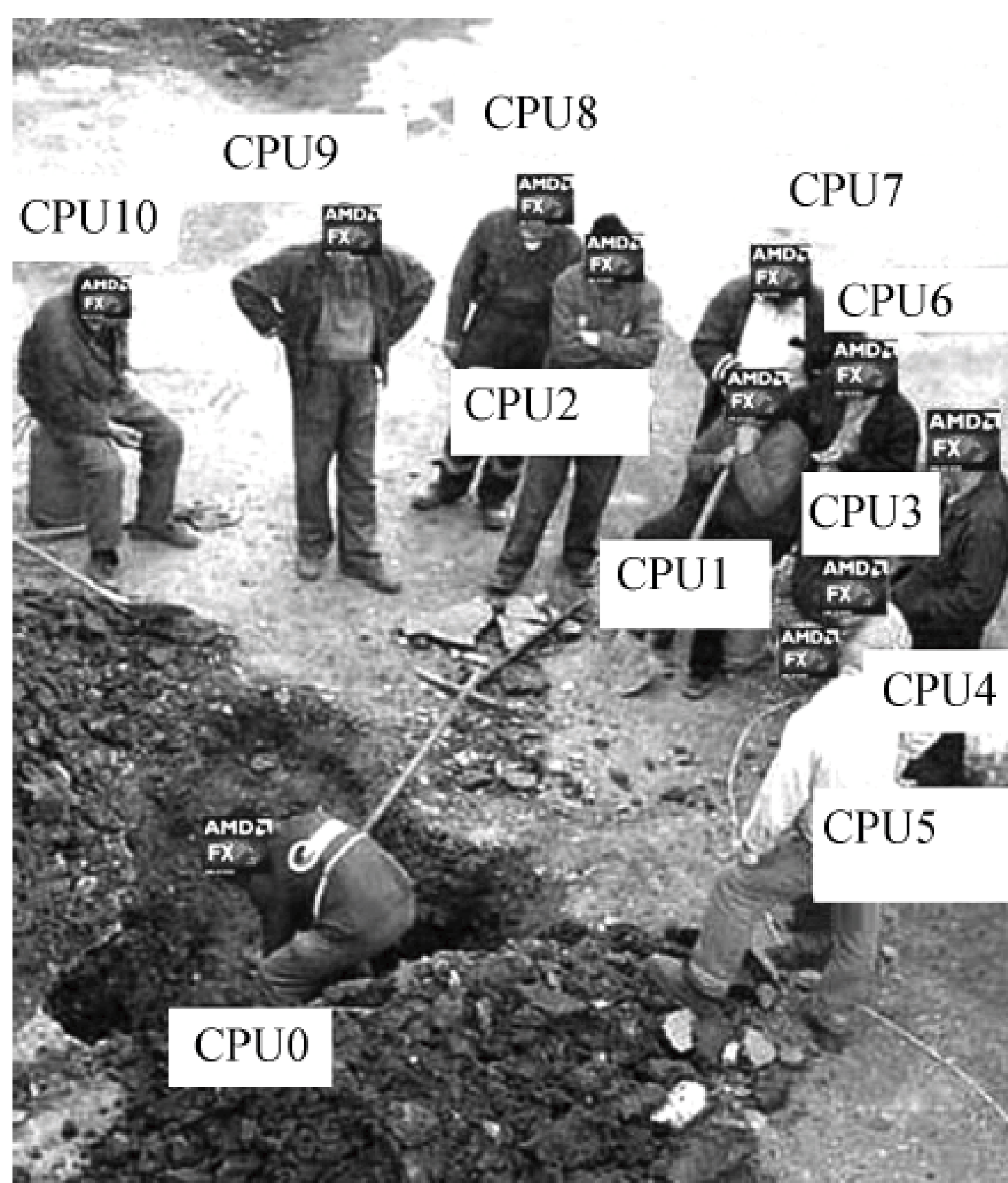


图 1－10　多路架构下负载不均衡的情况（图片来自 CSDN）

相对而言,多核单 CPU 在解决负载均衡问题上确实要好于单核多 CPU,毕竟 CPU 内部的核心通信效率很高,主存也可以共享。众核技术是在一个 CPU 中包含了多个核心的技术。与多核不同,众核一般指 CPU 中集成了 8 个以上的核心,因此也叫作 Network on Chip(NoC,片上网络)。众核技术多用于专用场景,比如防火墙、流处理、视频处理等,在大数据分析中也很有应用潜力。

多核技术一般采用片上网络/片内总线的方式互联各个核心,而众核一般采用 2D Full Mesh 网络来连接这些 CPU 核心,如图 1-11 所示。后者的路由是静态的,每个核心都有个 ID(其他组件也有 ID),并且 ID 与位置都是固定的,因此点对点之间的路径也是固定的,核间通信时带上目标核心的 ID 即可直接到达。

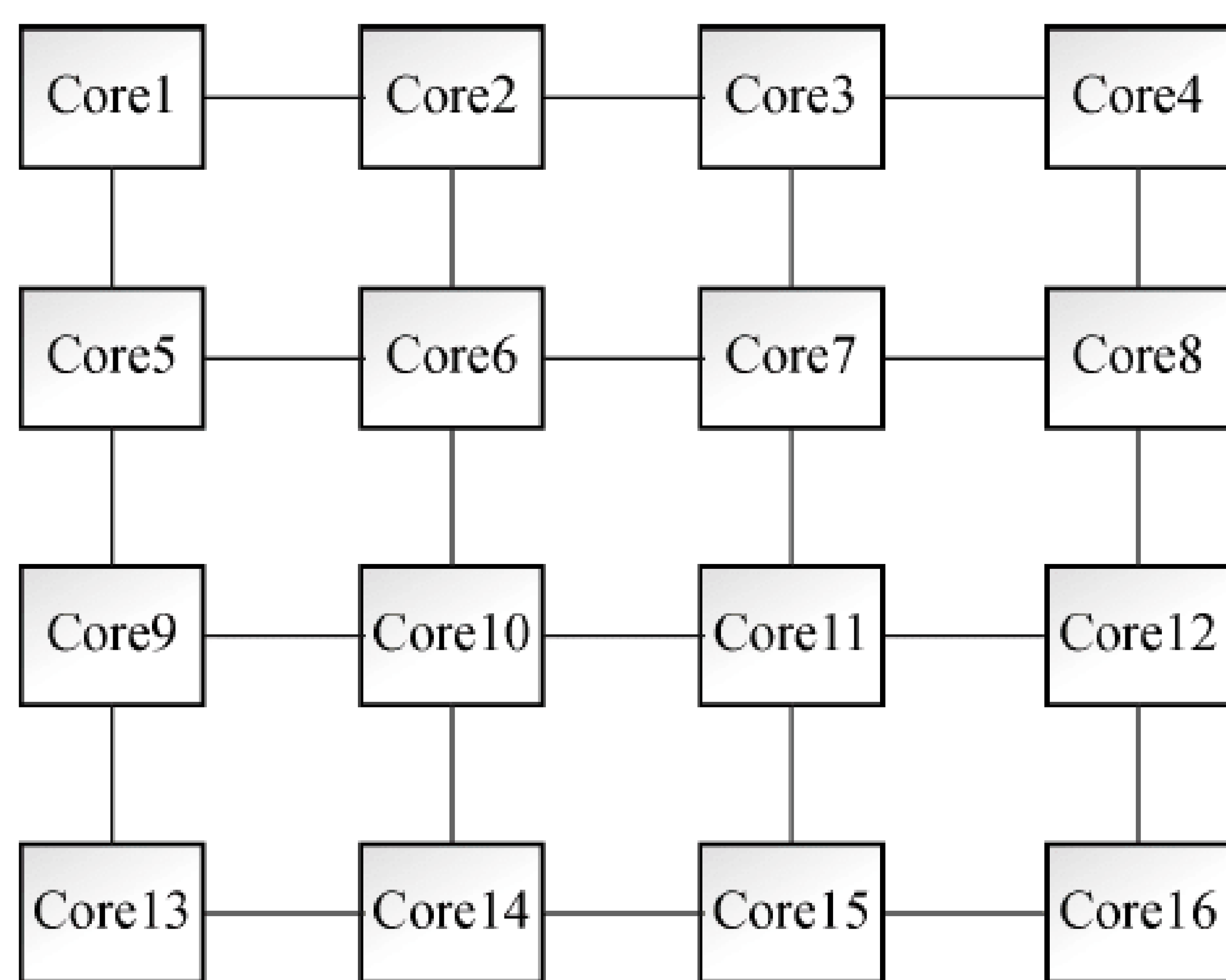


图 1-11　2D Full Mesh 网络组网示意图

众核的任务调度可分为对称式同构协作和非对称式异构协作两种方式。

- **对称式同构协作**:将待处理的任务分成多个切片,控制程序将这些切片推送到队列中,各个核心从队列中提取切片任务执行,执行完成后控制程序统一汇总输出结果。这种方式要求各个切片任务之间没有依赖关系。
- **非对称式异构协作**:要求每个核心处理不同数据的同一个工序,即流水线方式,与单指令多数据流(SIMD)技术类似。比如防火墙的网络包处理器 NP(Network Processor),每个工序只处理网络包协议栈的其中一层,每个核心处理不同网络包的同一个工序。

1.5　CPU 缓存机制

1.5.1　Cache 机制

Cache 就是 CPU 的缓存。对于 CPU 来说,虽然 DDR 技术的出现使得主存的存取速率大大加快,但对于同样也在不断以指数级提速的 CPU 来说仍然是太慢了,这一快一慢会使存取过程中 CPU 处于忙等状态。Cache 的出现就是为了弥补 CPU 与主存之间的读写速度的差





距,在中间做一个时间与空间的折中。那为啥不用寄存器?这是由于寄存器虽然快,但毕竟容量太小且价格太昂贵。而 Cache 的容量远大于寄存器,速度不是太慢,价格也不是太贵,因此就成了唯一选项。

前文讲过,CPU 的 Cache 一般分为三级,其中第一与第二级(L1、L2)缓存是集成到物理核心内部的,第三级(L3)缓存则一般放在物理核心之外。在整个存储系统的体系结构里,寄存器的存取速度无疑是最快的,但容量最小;L1 缓存的速度次之,容量大约有几十千字节;L2 缓存的速度又次之,容量大约有几百千字节;L3 缓存的容量最大,一般有数十兆字节,但速度也最慢,参见图 1-12。

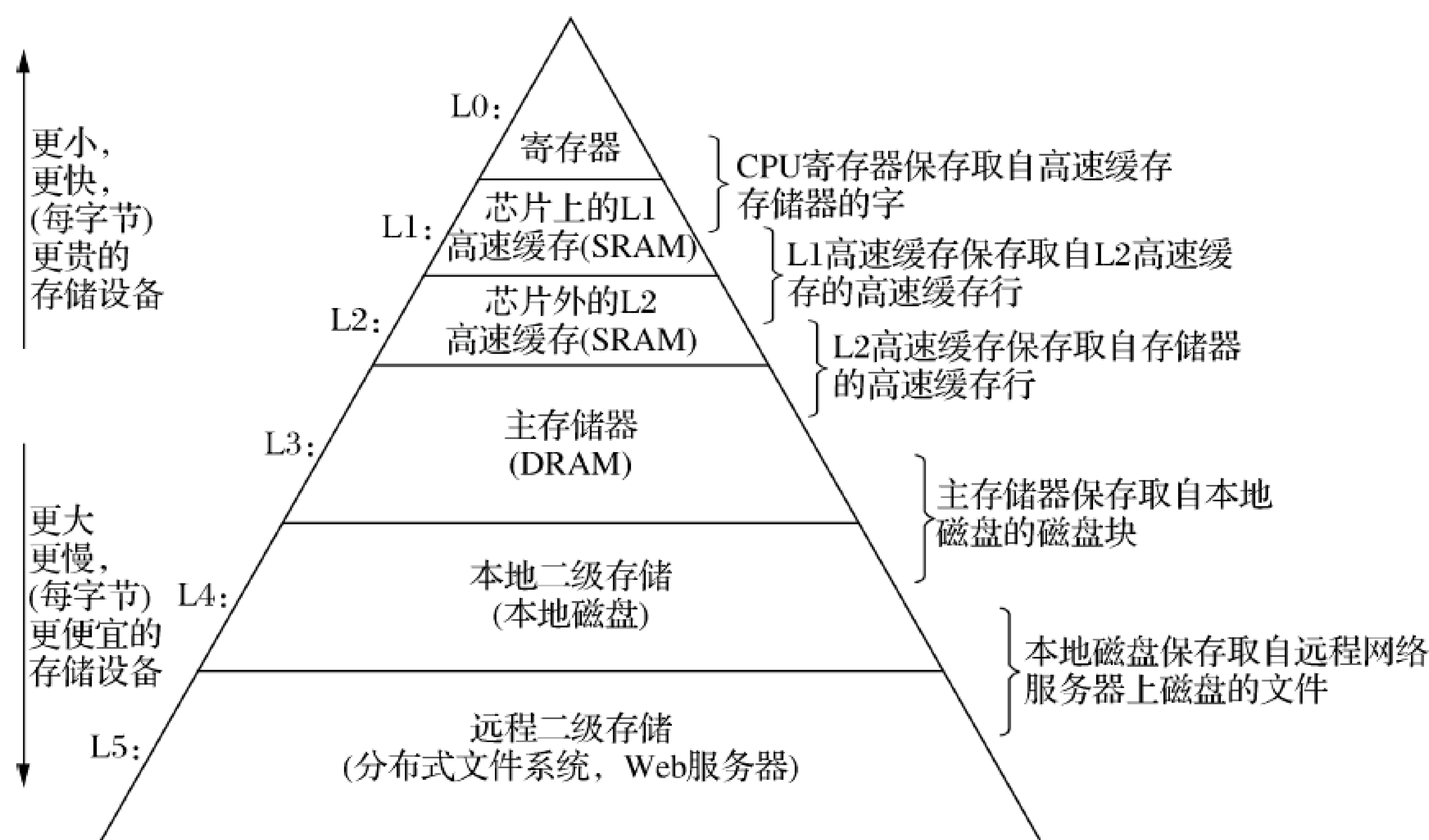


图 1-12　处理器各存储结构的存取速度

Cache 不是以内存页为单位与内存交换数据的,而是以一种叫作"Cache line"的结构为单位与主存交换数据,相当于缓存里面的"页面",也是 Cache 与主存之间传输的最小单位,一般一个 Cache line 为 64 字节大小,参见图 1-13。

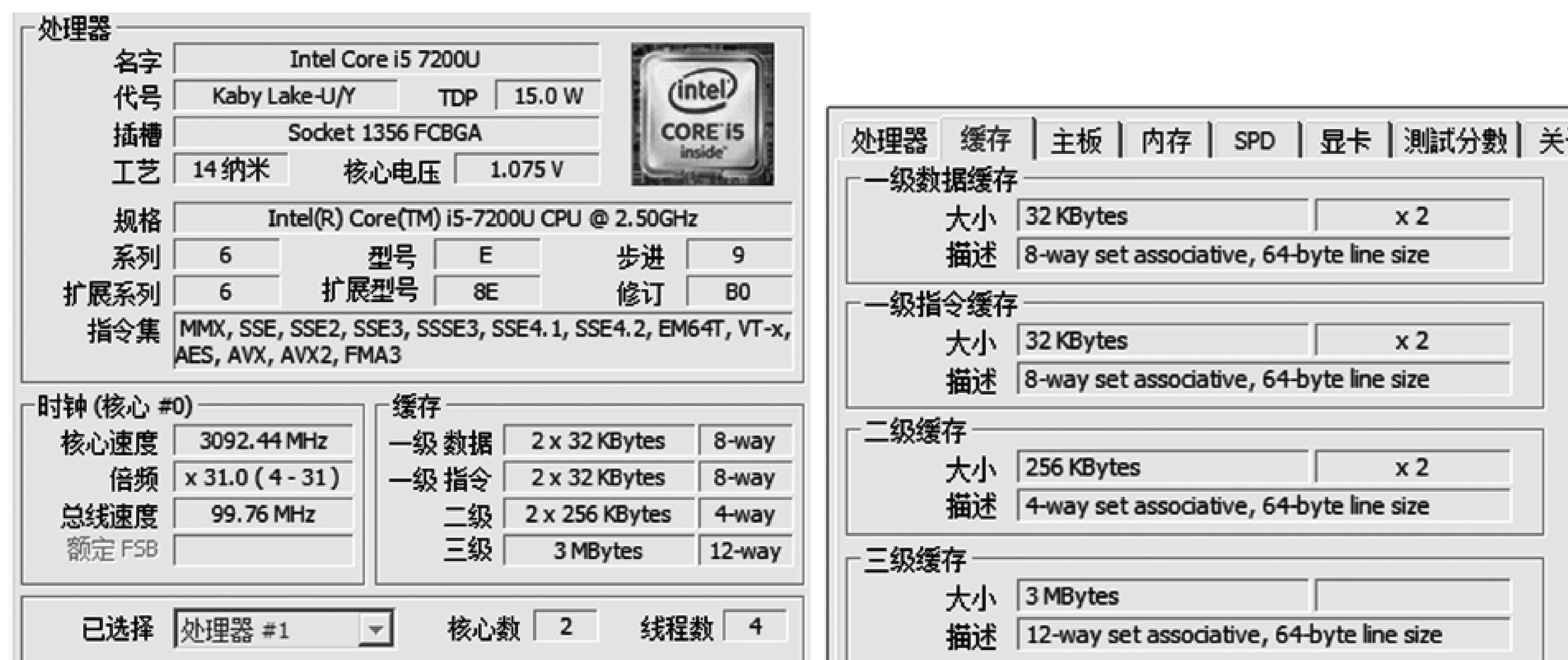


图 1-13　CPU 物理核心、逻辑核心与三级缓存

Cache 与主存交换数据也叫作“映射”，不过这里的交换只是单方面地把主存中的内容装入 Cache 中。将 Cache 中的数据写入主存叫作“写回”。当 CPU 要访问数据时会先在 Cache 中查找，找到了称为“命中”，命中后 CPU 将虚拟地址转换成 Cache 地址进行访问；否则称为“不命中”（Cache Miss），这时需要使用一定的映射机制将主存中的数据载入缓存中并提供给 CPU。在 NUMA 架构下，如果本地节点的 Cache 未命中，但跨越 QPI 的远程 Cache 被命中，则对于发起 Cache 访问的 CPU 节点来说仍然算作“LLC Cache Miss”（不命中）。

按照映射机制的不同，Cache 也相应地分成了三类：全关联型 Cache、直接关联型 Cache 和组关联型 Cache。

1.5.1.1　全关联型 Cache

在全关联型 Cache 中，主存中任何一块内存都可以以 Cache line 为单位映射到 Cache 的任意位置，这也是“全关联”这一叫法的由来，如图 1－14 所示。Cache 会为映射的数据建立目录表，每个表项由虚拟地址、Cache 块号和有效位三部分构成。

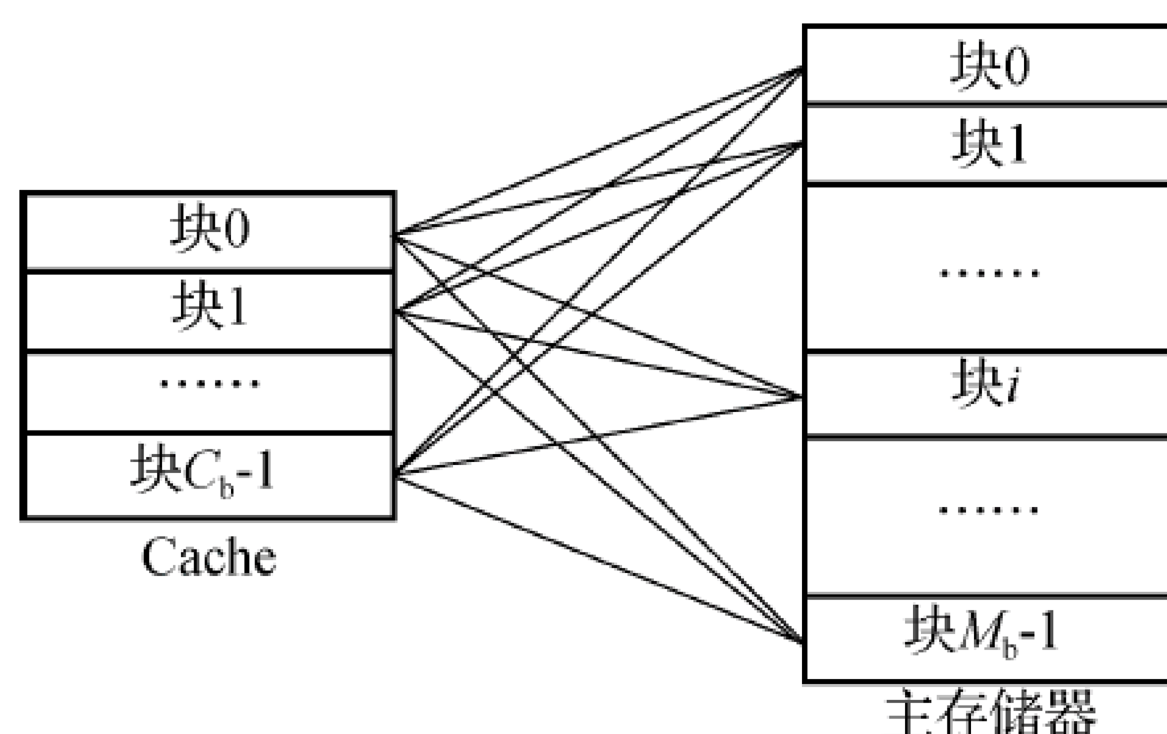


图 1－14　全关联型 Cache 的映射视图

CPU 访问某个虚拟地址时，首先通过 Cache 中的目录表查找目标内存是否存在于缓存中，存在则直接读取；不存在则从内存中映射。32 位系统下内存的虚拟地址也是 32 位的，被分成了三个逻辑部分（64 位系统的虚拟地址则分成了 5 部分，虚拟地址的概念我们在后面章节中会详细介绍）：页目录地址（Dircetory 部分）、页表地址（Table 部分）和页内偏移（Offset 部分），如图 1－15 和图 1－16 所示。

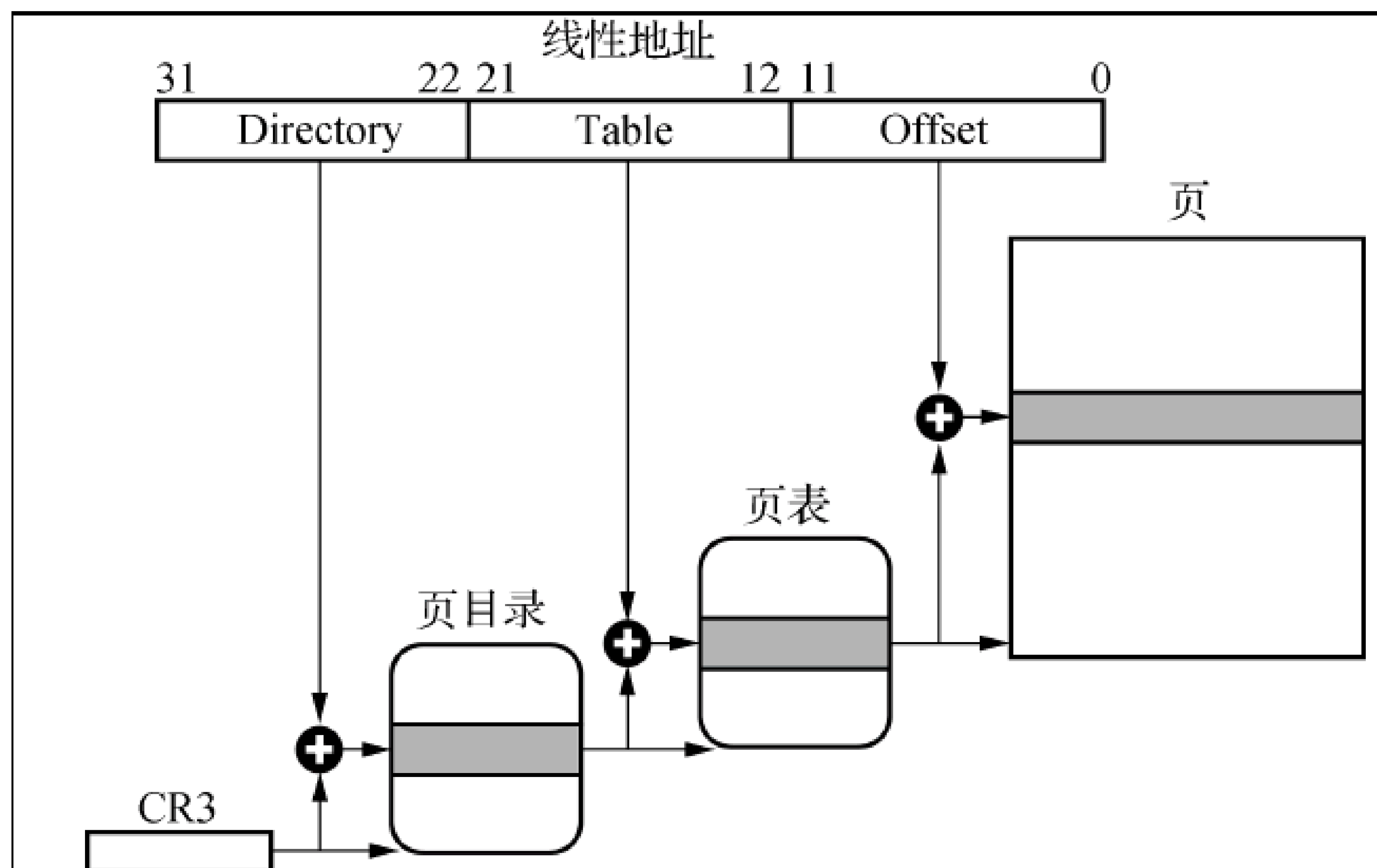


图 1－15　32 位系统下虚拟地址的三级寻址

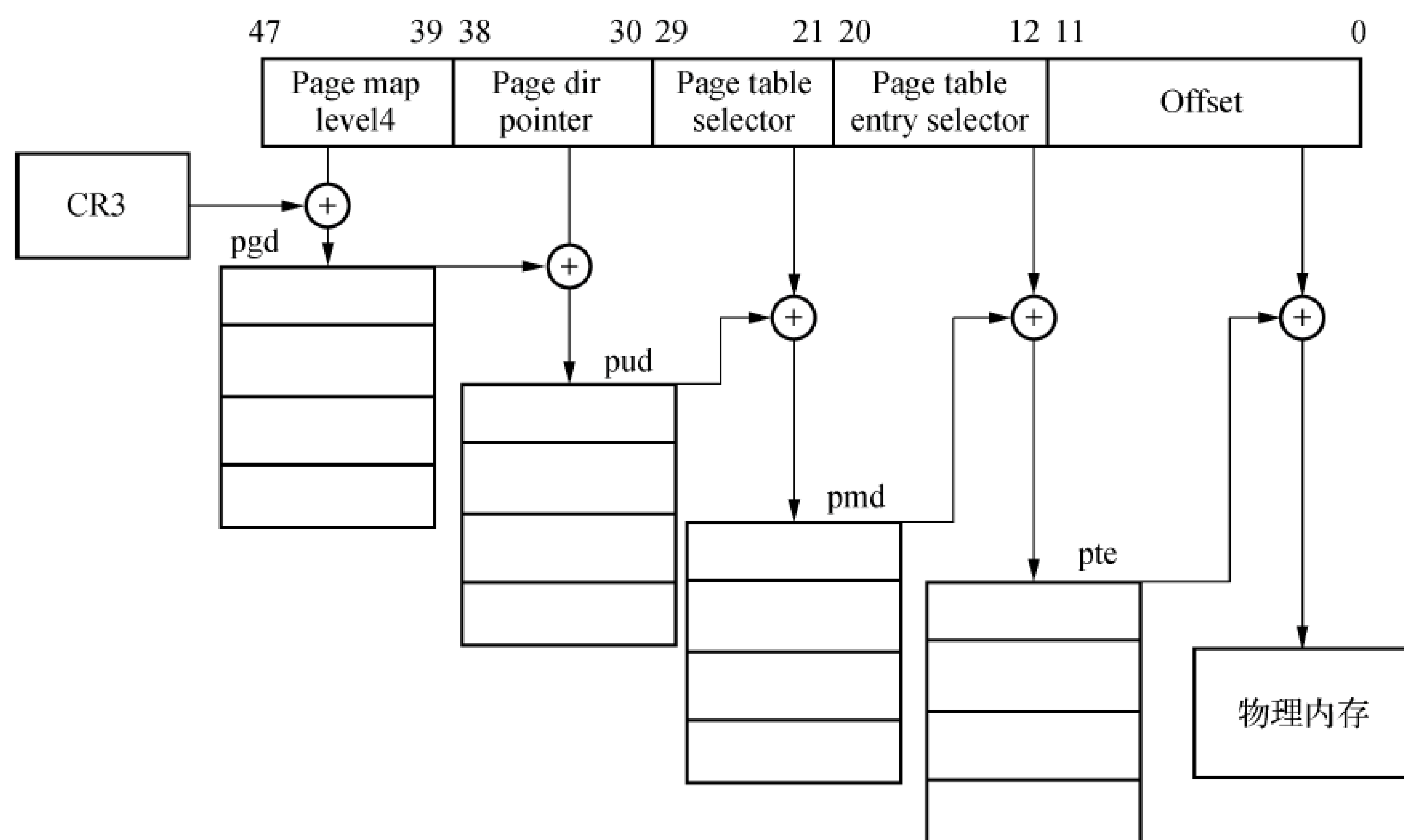


图 1 - 16　64 位系统下虚拟地址的五级寻址

在 32 位系统下全关联型 Cache 的虚拟地址可看作内存块(Cache line)块号 Tag 和块内地址 Offset 两部分,如图 1 - 17 所示。其中 Tag 是和 Cache line 对应的,而 Offset 即该内存单元在 Cache line 中的偏移,因此也可以推断出,Offset 占 5 位(一条 Cache line 占 64 字节),而 Tag 则占用 27 位。



图 1 - 17　全关联型 Cache 角度下的虚拟地址

CPU 在全关联型 Cache 中查找目录的过程如下:

① 首先在 Cache 的目录表中查找对应的 Cache line 地址;

② 若步骤①中找到对应的目录项,则检查目录项的有效位:有效则说明缓存命中,通过目录项中的内存块块号 Tag 找到 Cache 中对应的 Cache line,加上 Offset 找到相应数据位置;若有效位无效或直接未找到对应的目录项,则说明缓存未命中,CPU 转而在内存中映射对应的数据并刷新到 Cache 中。整个查找过程如图 1 - 18 所示。

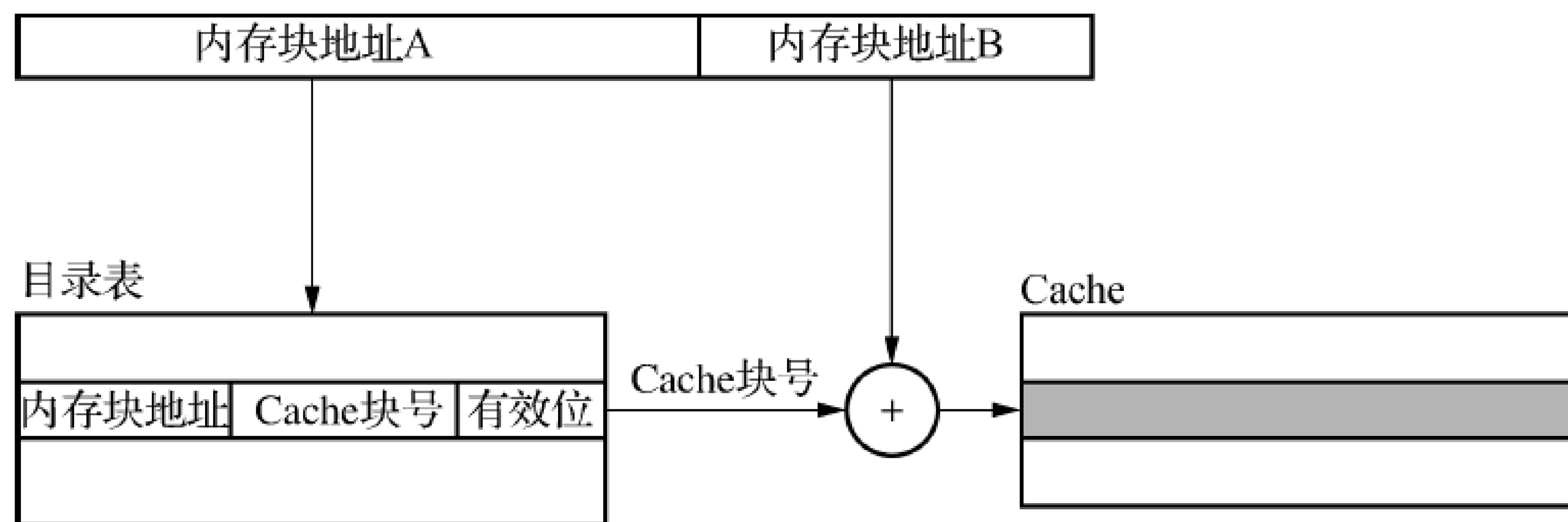


图 1 - 18　全关联型 Cache 的查找过程

1.5.1.2　直接关联型 Cache

直接关联型 Cache 将 Cache 分成了 N 条 Cache line，同时也将主存以 Cache 的容量为步长从低址到高址平均分成了 M 等份，主存中每一等份区域与 Cache 的容量大小一致。每一等份区域中的主存也分成 N 条 Cache line，主存中每一条 Cache line 与 Cache 中的 Cache line 对应，因此整个主存中有 M 条 Cache line 与 Cache 其中一条 Cache line 对应，如图 1－19 所示。直接关联型 Cache 虽然实现得比较死板，但也是一种简单和快速的方案。

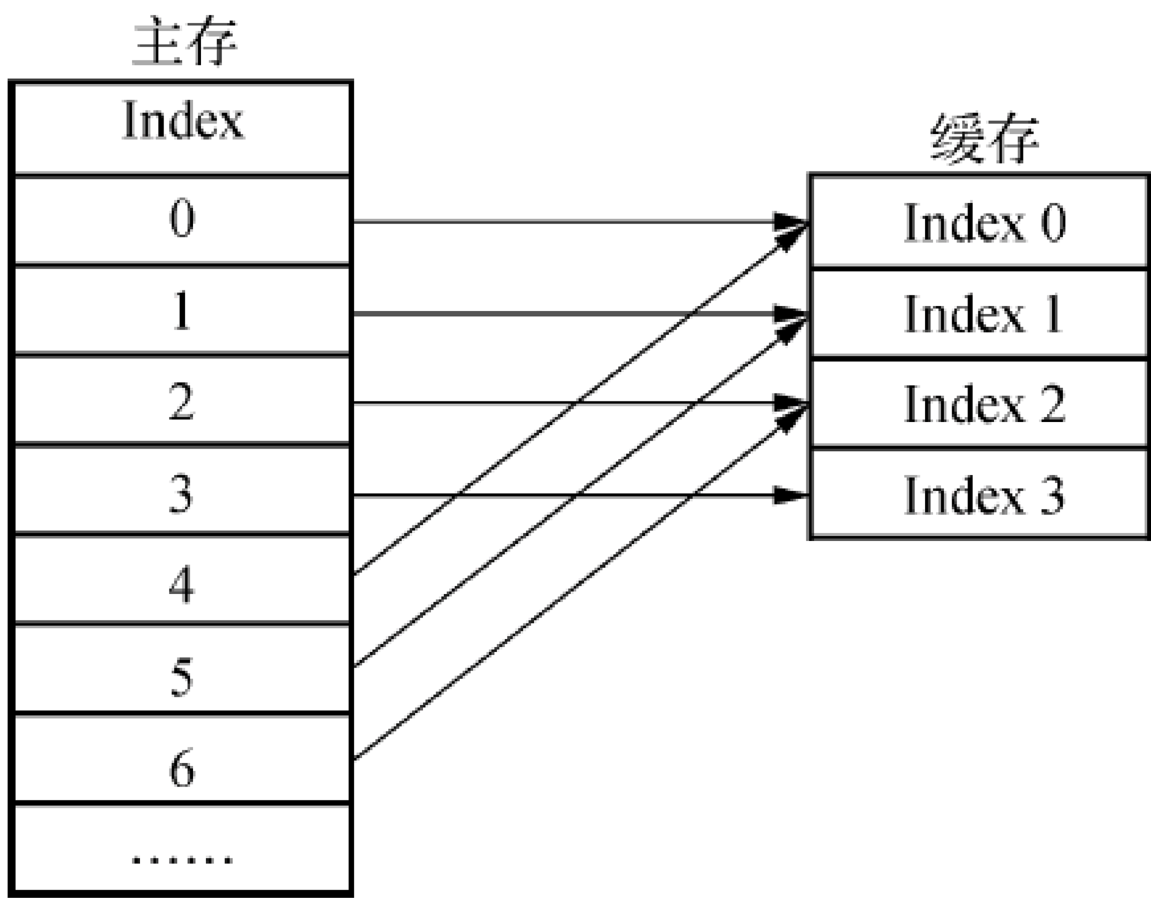


图 1－19　直接关联型 Cache 的映射视图

直接关联型 Cache 中目录表的表项由两部分组成：区号和有效位，而对应的主存中的虚拟地址则被分成了三个部分：区号 Tag、块号 Cache Set 和块内偏移 Offset，如图 1－20 所示。区号 Tag 表示主存 M 等份中的一份；块号 Cache Set 即每一等份中 Cache line 的序号（占用的位数与 Cache 的大小及 Cache line 的大小有关）；块内偏移 Offset 表示在一条 Cache line 中的偏移（占 5 位）。

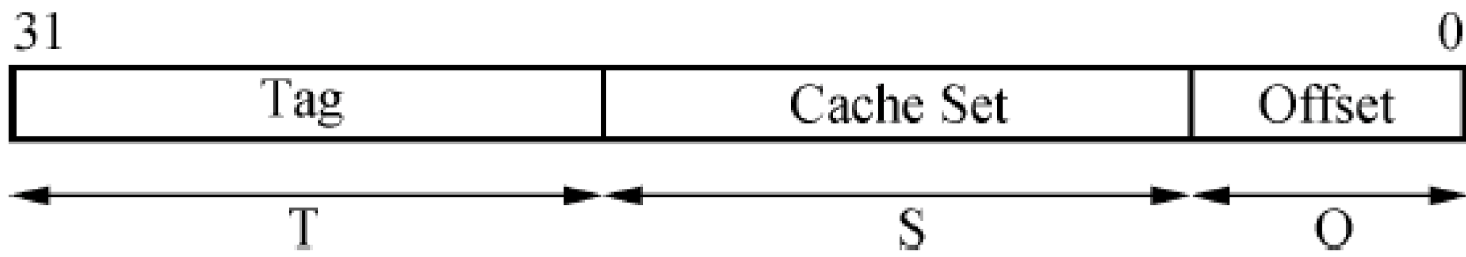


图 1－20　直接关联型 Cache 角度下的虚拟地址

CPU 在直接关联型 Cache 中查找目录的过程如下：

① 首先根据区号 Tag 在 Cache 的目录表中寻找对应的目录表项；

② 如果步骤①中找到了对应的表项，则检查有效位是否有效：若有效则根据块号 Cache Set（Cache line 的索引）找到 Cache 中对应的 Cache line，再与 Offset 累加即得目标内容的地址；若无效或找不到对应的表项则说明目标 Cache line 不在 Cache 中，需要从主存中映射。整个查找过程如图 1－21 所示。

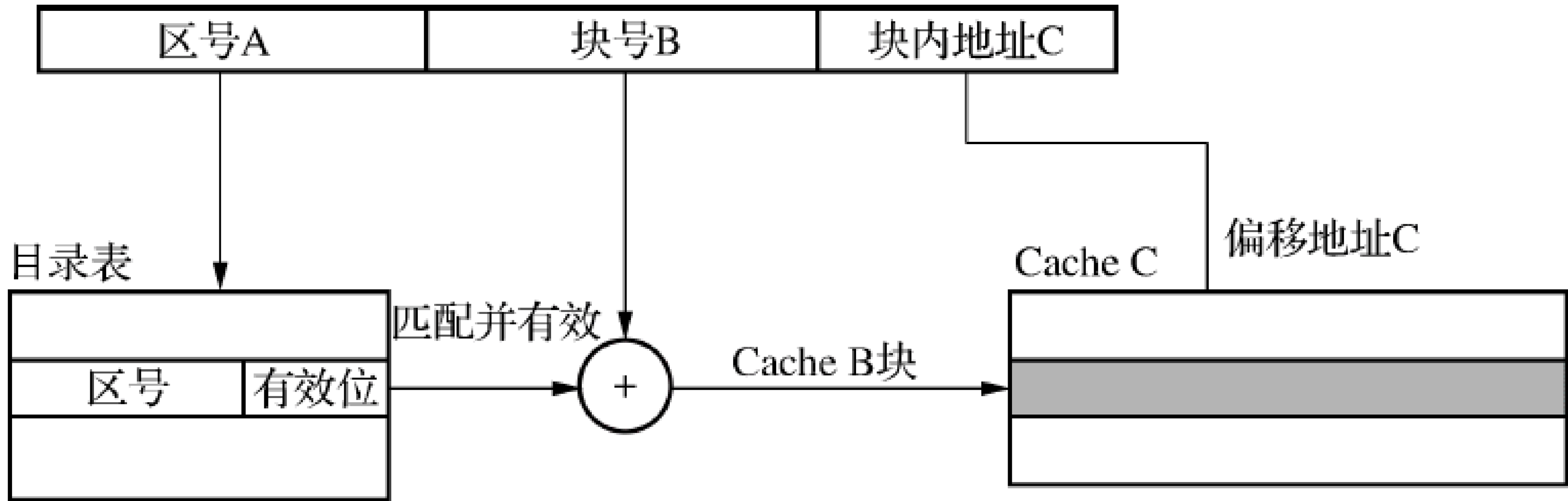


图 1－21　直接关联型 Cache 的查找过程

1.5.1.3　组关联型 Cache

组关联型 Cache 是上述两种方式的折中。在直接关联型 Cache 中，主存被分成了若干

等份，每个等份中块号相同的 Cache line 只能同时在 Cache 中出现一份，是“有他无我有我无他”的状态。但在实际应用中，这种映射方式并不是最高效的，因为为每一个等份区域分配的 Cache line 数量太少了（只有一条），所以往往导致 Cache line 频繁切换。组关联型 Cache 改善了上述状况，它将主存和 Cache 都分成了若干组，Cache 中的分组数量与主存中的每个组包含的 Cache line 块的数量相同，主存的 Cache line 块对应到 Cache 的哪个组是固定的，但对应到该组的哪一行是灵活的。如图 1－22 所示的是一个两路 Cache line 的分组方式（Cache 分组中有两条 Cache line）。

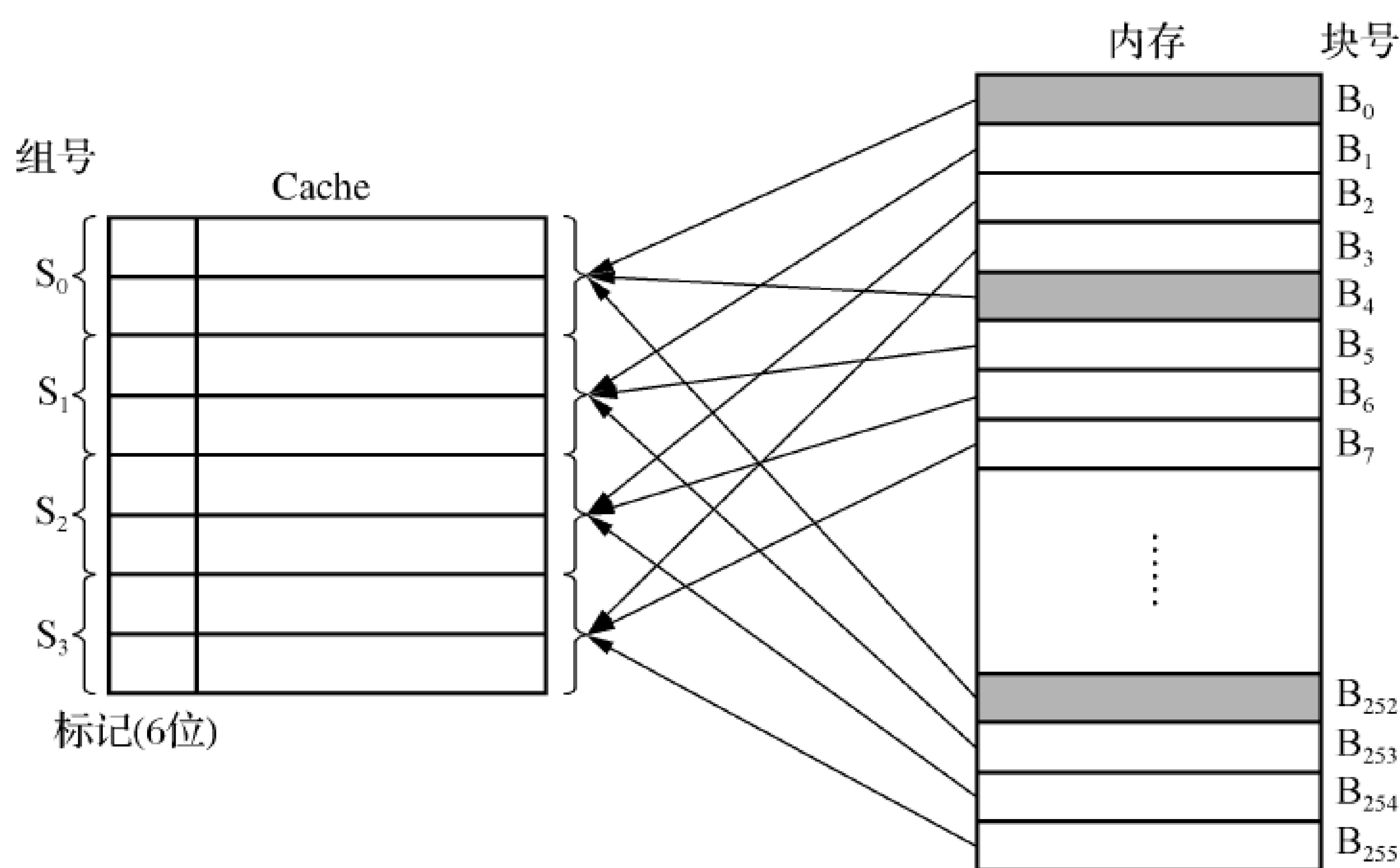


图 1－22　组关联型 Cache 的映射视图

组关联型 Cache 中的目录项被分为三部分，分别是区号＋块号、Cache 块号和有效位。而主存中的虚拟地址则相应地被分为了四部分：区号、组号、Cache 块号和块内偏移，其中区号＋块号决定了在 Cache 中的 Cache line 的索引。CPU 在 Cache 中查找目录项的过程如下：

① 根据虚拟地址中的区号＋块号在 Cache 的目录表中查找；

② 如果步骤①中找到了对应的区号＋块号，则检查有效位是否有效：若有效，则表明目标内容在 Cache 中，根据组号找到 Cache 中的 Cache line，再加上偏移就是目标内容在 Cache 中的地址；否则，就要到内存中映射读取。整个查找过程如图 1－23 所示。

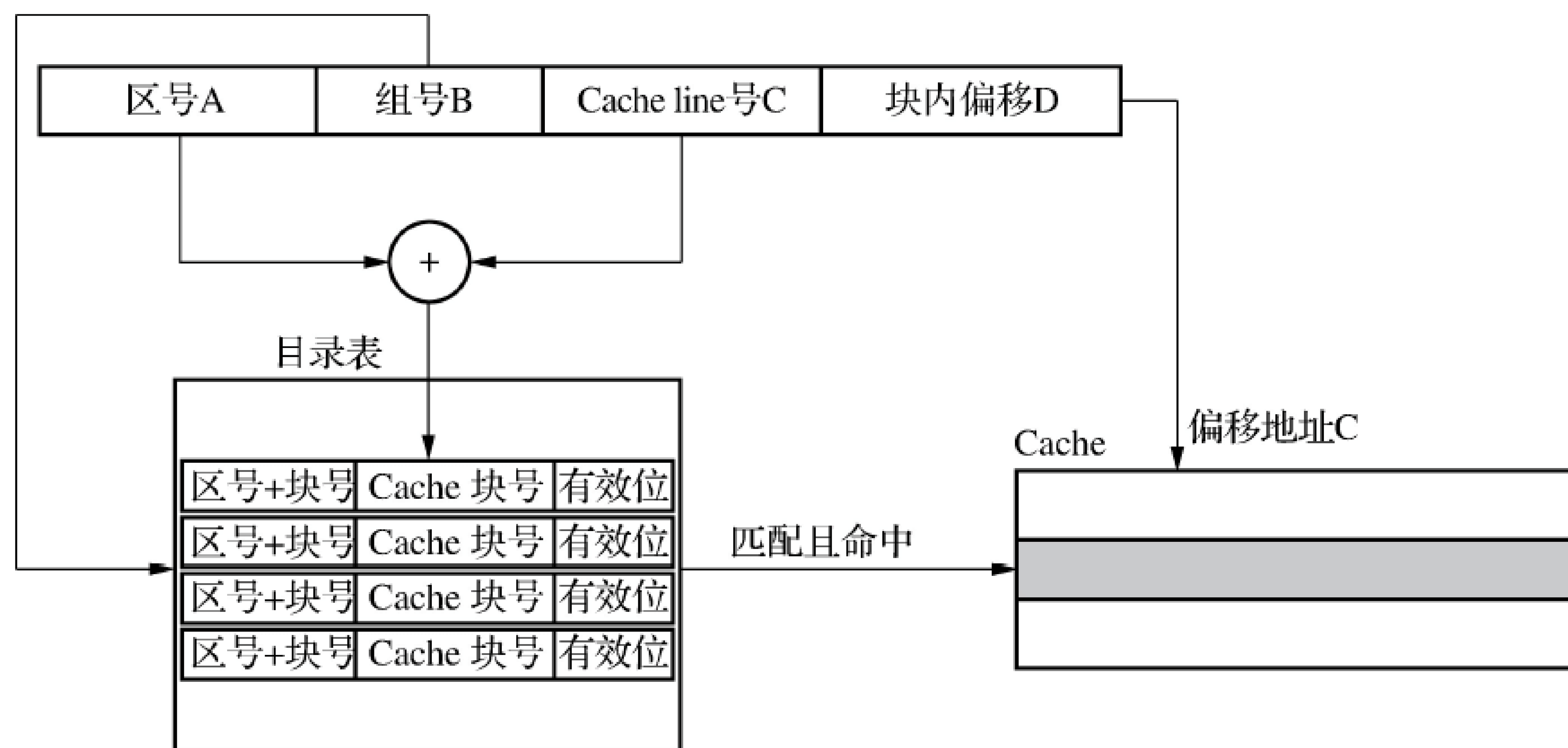


图 1－23　组关联型 Cache 的查找过程

1.5.1.4　存在的问题

1. Cache 一致性问题

如果多个处理器同时对缓存中某个相同地址的内存块(Cache line)进行读写就会引起 Cache 一致性问题,本质上这是因为每个 CPU 会独享各自的缓存,且更改缓存后不会立即“落盘”记录到主存,导致主存中的数据在各个缓存中都有不同的备份(单路单核处理器不会出现 Cache 一致性问题)。

一致性问题的解决方案有两种协议机制:基于目录的协议(Directory-Based Protocol)和总线窥探协议(Bus Snooping Protocol)。

- **基于目录的协议**:缓存在 Cache 中的内存块需要将地址统一记录在一个全局目录表中,以此协调一致性问题。当 CPU 需要将某块数据从内存加载到独享 Cache 中时要向目录表提出申请;当 Cache 中的内存块被改变时,目录表也要改变其状态,更新其他 CPU 的 Cache 备份。基于目录的协议是一种全局统一管理式方案,其实时性较总线窥探协议稍低,但适用于大规模多处理器系统。
- **总线窥探协议**:被 CPU 独享的 Cache 中的内容一旦被本地处理器改变,则需要通过总线广播;每个 CPU 都会监听总线,一旦监听到了改变通知则改变本地 Cache 中相应数据的备份。总线窥探协议是一种分布式协商通知式方案,其实时性较基于目录的协议更高,适用于小规模多处理器系统。

一般的处理器都实现了 Cache 一致性协议,以总线窥探协议居多,而总线窥探协议又以 MESI 协议与 MESIF 协议为代表。在 MESI 协议中,每个 Cache line 都有两个标志:dirty(数据是否被修改)标志和 valid(数据是否有效)标志,描述了 Cache 和主存之间的数据关系。MESI 是 Cache line 的 4 种状态的首字母缩写,详见表 1-2。

表 1-2　Cache line 的 4 种状态及其迁移方向

当前状态	触发事件	解释	迁移状态
修改态(M)	总线读	侦测到总线上有其他处理器在请求读该行,刷新该行至内存,以便其他处理器能用到最新的数据,并且状态更新为 S 态	S
	总线写	侦测到总线上有其他处理器请求“意图”写该行,即请求独占态,刷新该行至内存,并且设置本地副本为 I 态	I
	处理器读	本地处理器对该行进行读操作,不改变状态	M
	处理器写	本地处理器对该行进行写操作,不改变状态	M
独占态(E)	总线读	侦测到总线上有其他处理器请求读该行,因为本地处理器还没有对该行进行写操作,因此缓存内容与内存中内容一致,仅仅改变成 S 态	S
	总线写	侦测到总线上有其他处理器请求“意图”写该行,即另外有处理器请求独占该行,并且有写的意图,因此设置成 I 态	I

续表 1－2

当前状态	触发事件	解释	迁移状态
独占态(E)	处理器读	本地处理器对该行进行读操作,不改变状态	E
	处理器写	本地处理器对该行进行写操作,不改变状态	M
共享态(S)	总线读	侦测到总线上有其他处理器请求读该行,不改变状态	S
	总线写	侦测到总线上有其他处理器请求“意图”写该行,进入 I 态	I
	处理器读	本地处理器对该行进行读操作,不改变状态	S
	处理器写	本地处理器对该行进行写操作,不改变状态	M
无效态(I)	总线读	侦测到总线上有其他处理器请求读该行,不改变状态	I
	总线写	侦测到总线上有其他处理器请求“意图”写该行,不改变状态	I
	处理器读	Cache 不命中,产生一个读请求,送到总线上,内存数据到达 Cache 后进入 S 态	S
	处理器写	Cache 不命中,产生一个“意图”写该行的信号到总线,然后进入 M 态	M

由此也引出了著名的 MESI 定律:在所有的脏缓存段(M 状态)被回写后,任意缓存级别的所有缓存段中的内容和它们对应的内存中的内容一致。此外,在任意时刻,当某个位置的内存被一个处理器加载到独占缓存段(E 状态)时就不会再出现在其他任何处理器的缓存中了。

2. Cache line 伪共享问题

一个 Cache line 是可以被多个线程所使用的。各线程中有些变量虽是不同的,但却实际上存储于同一条 Cache line 中。如果有某线程修改了其中一个变量的值,其他线程可能会强制重新映射 Cache line。这是因为一致性协议是以 Cache line,如图 1－24 所示,而不是以单个独立的变量元素为单位的。这种大动干戈的数据共享方式称作“伪共享”(False Sharing)。

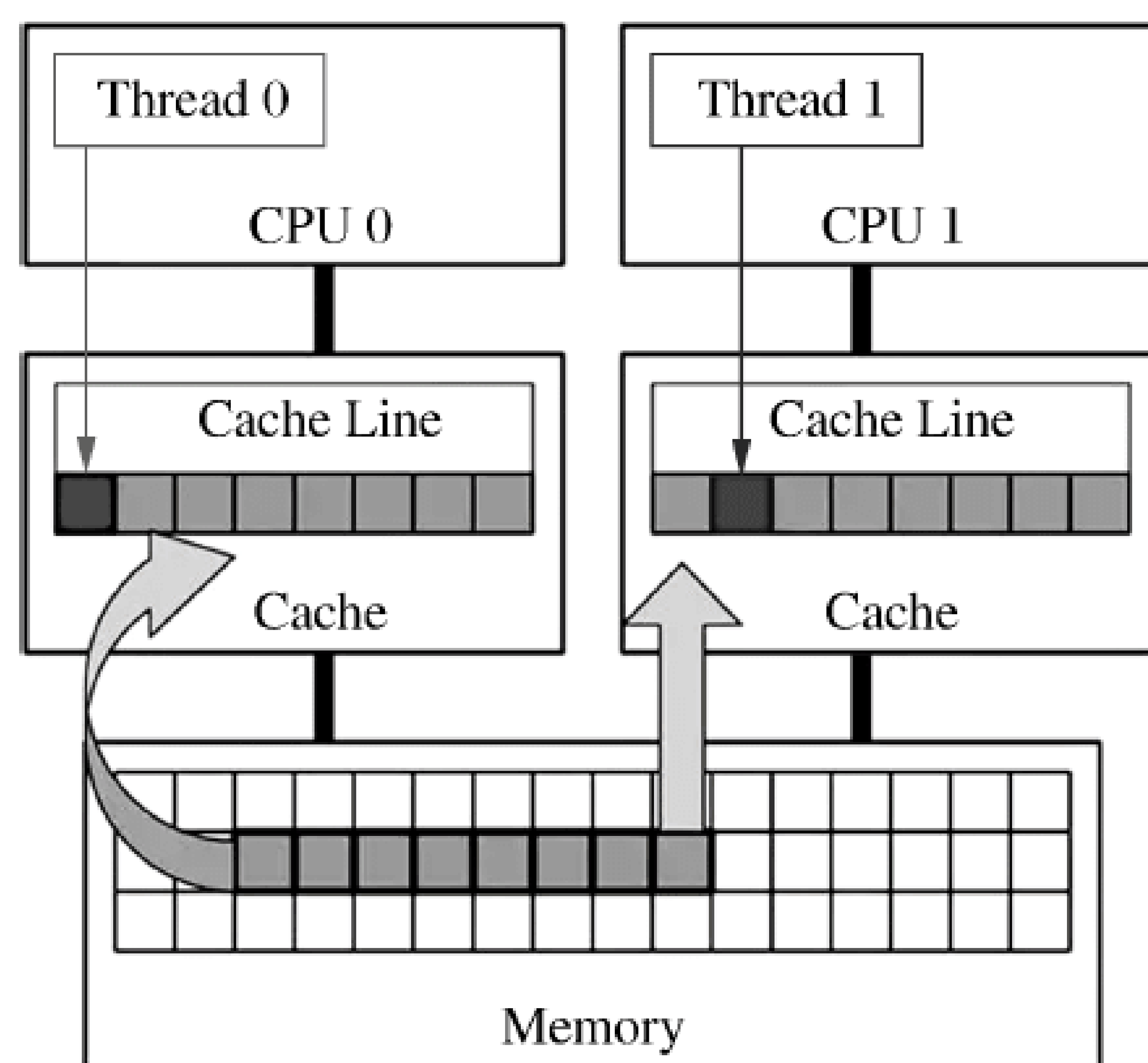


图 1－24　Cache line 的伪共享问题

解决伪共享问题可以采用如下方案：增大变量元素的地址间隔以使不同线程存取的元素位于不同的 Cache line 上。这是典型的以空间换时间的方案。

1.5.2　TLB

CPU 不仅有上述三种 Cache，还有一种特殊的缓存部件：TLB（Translation Look-aside Buffer，旁路转换缓冲，或称为快表），用于缓存内存中的页表项。TLB 本质上就是一种 Cache，其存储介质也是 SRAM，只不过存放的内容与上述几种 Cache 不同，TLB 中存放的是内存的页表项（关于页表与页表项的内容在后续章节中会有详细介绍）。TLB 一般采用相连存储器的方案，用虚拟地址进行搜索而返回对应的物理地址，以此来避免需要多次访问内存才能得到物理地址的延迟问题。如果没有 TLB，通过虚拟地址访问物理地址要经过三段式查表（32 位系统），即先从虚拟地址高位部分获取页目录表的索引继而得到页目录表项，再通过虚拟地址中位部分获取页表索引继而得到页表项，最后通过虚拟地址低位部分获取内存页的内容偏移。从过程来看实在是过于烦琐与耗时。

TLB 保存了虚拟地址的高 20 位与页帧号的对应关系，CPU 查找虚拟地址时先到 TLB 中匹配高 20 位，如果能够匹配成功（命中）则可直接获取物理页帧号，但如果不命中则仍然需要进行三段式查找，如图 1－25 所示。

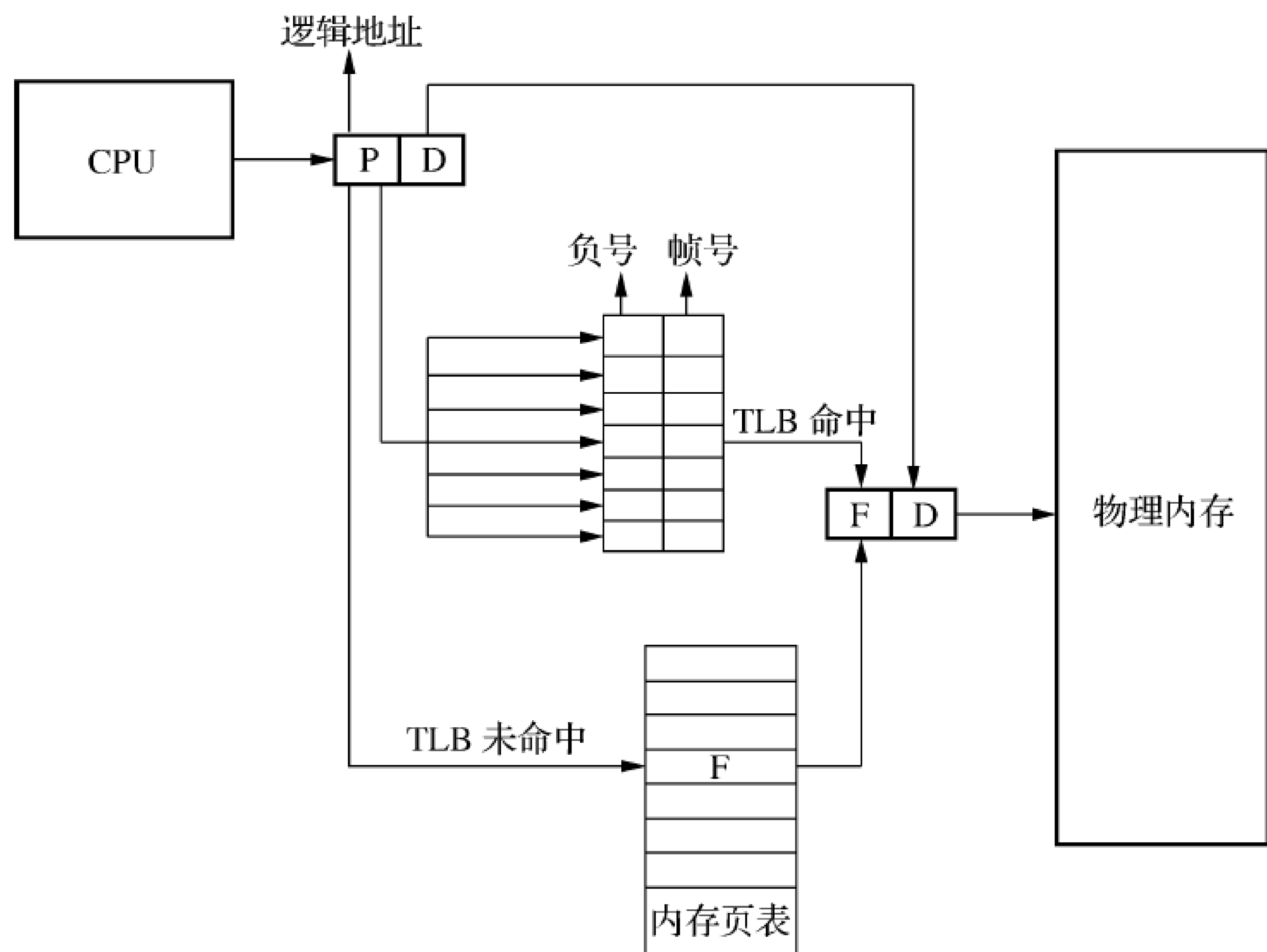


图 1－25　TLB 查找过程

有些处理器为了提高处理效率还将 TLB 进行了分组，一般分为 4 组，分别用于存储下列内容：

- 缓存常规页表（4 KB 页）的指令页表项（Instruction-TLB）；
- 缓存常规页表（4 KB 页）的数据页表项（Data-TLB）；
- 缓存大尺寸页表（4 MB 页）的指令页表项（Instruction-TLB）；

➢ 缓存大尺寸页表(4 KB 页)的数据页表项(Data-TLB)。

1.6 国产 X86 架构 CPU 现状

X86/X64 架构是 PC、服务器和超算领域处理器的主流架构,除了具有良好的计算性能和扩展加速能力,还囊括了对多数操作系统的支持,也兼容了多种异构扩展芯片,具有良好的生态。其他架构的 CPU 都不具备与 X86/X64 正面对抗的能力和实力,更得不到全生态的支持,无法挑战由微软和 Intel 构筑的事实上的"Wintel"联盟。

国产 X86 架构处理器的技术来源主要有两个:AMD 和 VIA(台湾威盛),分别向天津海光(HYGON)和上海兆芯授权 X86 架构,前者主要面向高性能服务器的处理器市场,后者定位于桌面级处理器市场。

1. 海光 X86 CPU

X86 架构是 Intel 的知识产权,虽然有交叉授权条款存在,AMD 也固然可以使用 X86 架构的专利技术,但如果 AMD 要对第三方授权 X86 架构全套指令集也不是一厢情愿的事。虽然 AMD 无权将 X86 指令集的专利技术授权给第三方,但是其自主设计的 X86 CPU 架构是可以被授权的,也就是说不能授权的是 X86 的基础专利技术,但是基于 X86 基础技术衍生出的公版 CPU 架构则属于 AMD 自主持有,是可以授权给第三方的,这是不违反交叉授权协议的。

而现实也的确如此。AMD 向海光授权的是已经设计好的 CPU 内核,或者说就是 AMD 基于 X86 基础技术自研的 Zen 内核,可能还包括了将不同功能的芯片集成到同一片上系统(SoC)的相关技术,这是异构计算架构的核心技术。但是 Zen 内核仅仅是 CPU 内核而不是 X86 指令集,获得方无法对 CPU 进行设计改进,也就无法衍生出自主知识产权的 CPU 架构。当然 Zen 的全套设计还是可以看的,但光是"看"能不能为国产 CPU 的设计带来实质性技术进步还要两说。

2. 兆芯 X86 CPU

兆芯的 X86 技术来源于 VIA,也是基于与海光一样的技术引进道路。2014 年兆芯推出了仿制的 X86 处理器 ZX-A,2016 年其自主设计的 ZX-C 四核处理器实现量产,但内核还是 VIA 的 Isaiah 2,采用的是 28 nm 工艺,兼容 Windows、Ubuntu 以及中科方德、中标麒麟、普华等国产操作系统。

2017 年底,兆芯发布了自主设计的 ZX-D 处理器(基于自主设计的"五道口"处理器架构),该型号处理器面向便携式设备和桌面设备。ZX-D 处理器采用了全新内核,最多支持 8 个核心,IPC 性能比上代提升 25%,单芯片性能提升 140%,内存带宽提升 120%;基于业界先进的 SoC 架构,并且整合了高性能集成显卡,兼容 X86 32/64 位指令集、SSE4.2/AVX 指令集,支持双通道 DDR4 内存、CPU 虚拟化、SM3/SM4 国密算法以及 9 个 PCI-E 3.0 端口等。

2019 年 6 月,ZX-E 系列处理器发布,官方宣称为国产首款主频达 3.0 GHz 的通用处理

器,采用的是 8 核 16 nm 工艺,其单颗 SoC 芯片包含了 CPU、GPU 和芯片组,具备高性能和低功耗特点,适合 PC、超级本、服务器和嵌入式计算平台等各种硬件平台,性能也相当于 Inter 目前主流的第 7 代 i5-7400 的水平。

另外值得一提的是,在技术上兆芯是国内唯一的拥有 CPU、GPU、芯片组三种 IP 的厂商。

本章小结

本章总结了处理器的相关体系架构,首先梳理与之相关的名词和概念,解释了精简指令集和复杂指令集的含义以及它们都包含了哪些架构的处理器,继而介绍了统一内存存取(UMA)架构、非统一内存存取(NUMA)架构和众核体系架构的概念。

同时,本章节也介绍了处理器的三级缓存(Cache)结构以及访问缓存的原理,并简单描述了 Cache 的一致性和伪共享问题。

最后还介绍了国产 X86/X64 处理器的现状,为后文介绍自主可控专题埋下了伏笔。

第2章 Windows 整体框架

任何软件要想编译或运行,离开了操作系统的支持是不可想象的。在各种操作系统中,无论是易用性、性能、安全性还是普及程度,Windows 都处于一骑绝尘的优势地位。虽然 Windows 系统不开源,但是业内众多高手纷纷通过软件调试和逆向工程等方法,或全局或微观地还原了 Windows 的一些全貌和细节。我们研究软件技术、安全技术、逆向技术、传输技术等就不得不研究 Windows 系统。

本章首先介绍 Windows 的发展历程,继而按照图 2-1 所示的提纲讲述 Windows 整体系统框架,希望能为业内同行提供一些技术参考和原理分析。

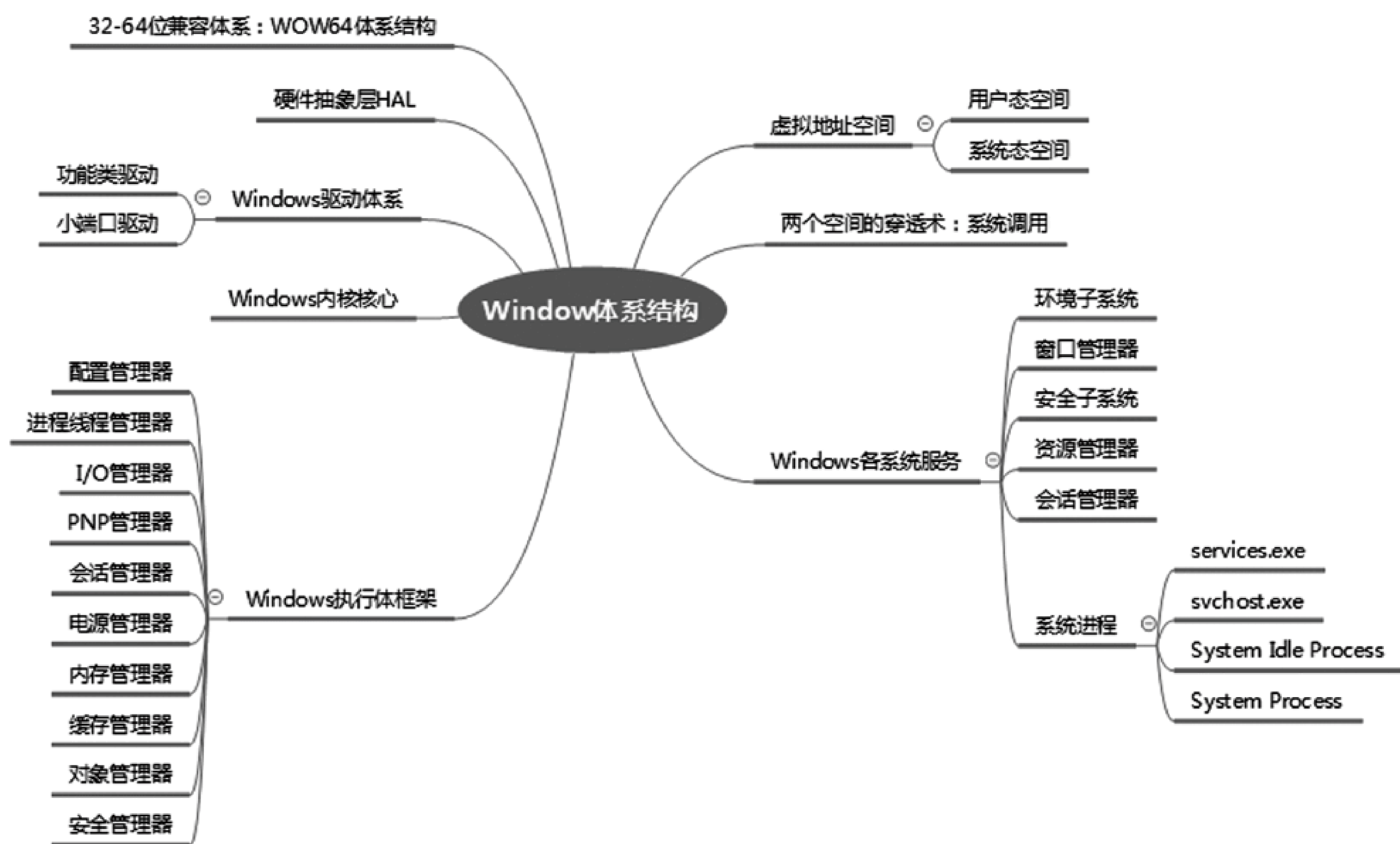


图 2-1 本章提纲

2.1 Windows 操作系统的历史

Windows 操作系统大致可以分为桌面操作系统、服务器操作系统、嵌入式系统(Windows CE)三类。在此我们只分析桌面操作系统和服务器操作系统。桌面操作系统和服务器操作系统各有所长,例如桌面操作系统可能会加入媒体管理、3D 渲染加速等模块以服务于对于流媒体、游戏性能有要求的场景,CPU 时间片更短一些,以利于用户操作的实时性要求;而服务器操作系统在这方面可能就弱一些,在 CPU 时间片的分布上,服务器操作系统的时间片

更长一些,这是因为后台服务/进程对于操作的实时性要求不是那么高,但也不希望发生频繁的上下文切换,同时服务器操作系统也支持更大规模的多处理器架构。

Windows 系统的发展历程如图 2－2 所示,对于桌面操作系统,2000 年发布的 Windows 2000 是一个里程碑式的存在,其也被称为 Window NT 5.0,代表了 NT 5.0 内核。从那时起到现在的 Windows 系统内核的机制与架构基本没有什么根本性改变。表 2－1 是 Windows 2000 系统以后的版本演进细节。

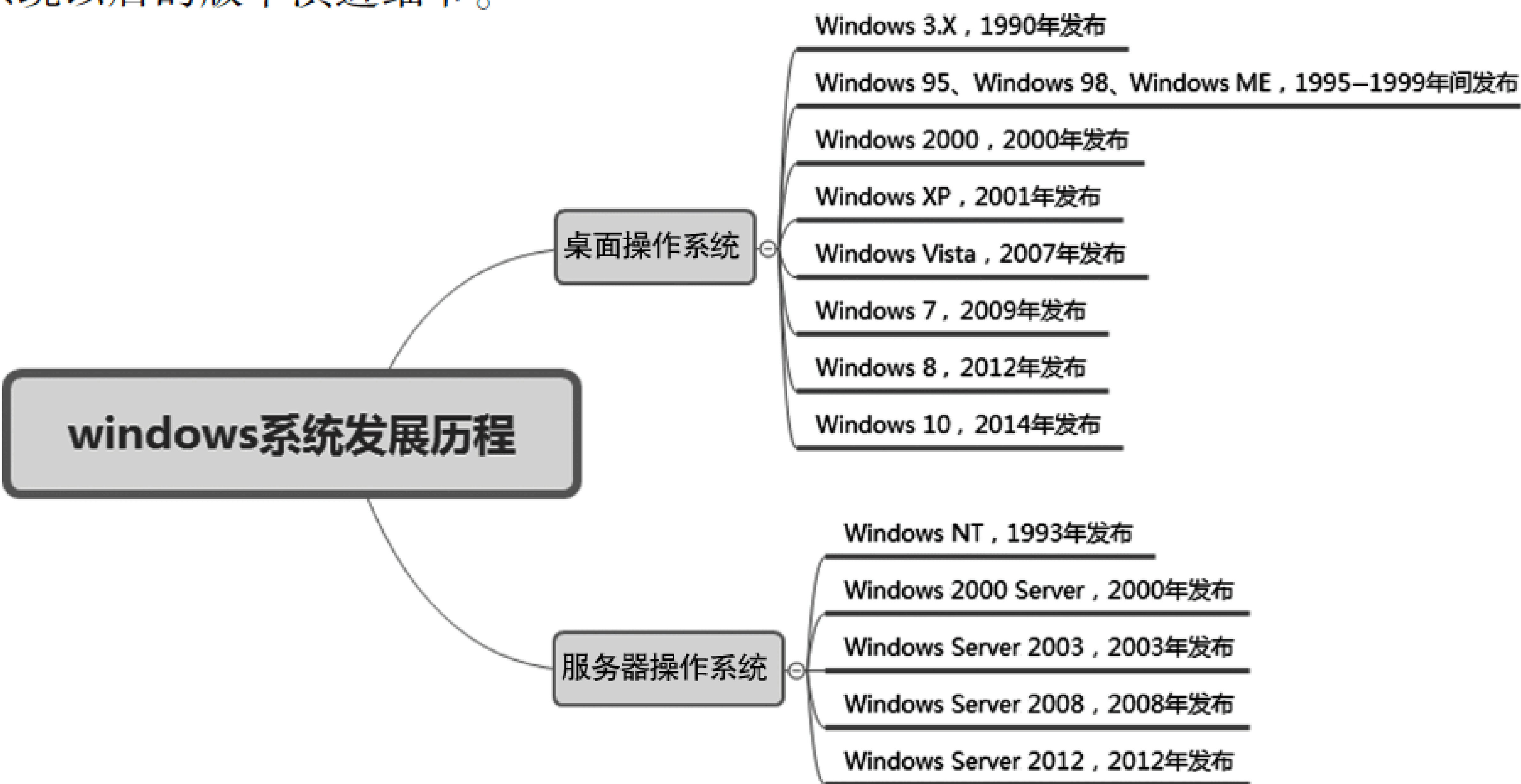


图 2－2　Windows 操作系统发展历程

表 2－1　Windows 操作系统各版本主要特性

NT 版本号	操作系统名称	主要特性
NT 5.0	Windows 2000	商业性质的 32 位图形操作系统,也是从这个版本开始,微软推出了基于 NT 内核的桌面系统
NT 5.1	Windows XP	全新的桌面感观,分为家庭版和专业版两个版本
NT 5.2	Windows XP 64 位版本	微软第一个 64 位桌面操作系统
	Windows Server 2003	改进的 Active Directory(活动目录)(如可以从 schema 中删除类)、改进的 Group Policy(组策略)操作和管理以及改进的磁盘管理,如可以从 Shadow Copy(卷影复制)中备份文件,分为 WEB 版、标准版、企业版、数据中心版
	Windows Server 2003 R2	Windows Server 2003 的改进版本,包含了诸如增强的分布式文件系统命名空间管理界面等新增功能
NT 6.0	Windows Vista	更好的安全性,众多新功能,对操作系统核心进行了修正
	Windows Server 2008	代表了下一代的 Windows Server,对服务器和网络基础设施的控制能力更强,更加注重安全性和网络保护
NT 6.1	Windows 7	不需驱动就使用触控技术的 Windows 桌面操作系统,也集成了 DirectX 11 和 Internet Explorer 8,其中 DirectX 11 增加了新的计算 shader 技术,可以允许 GPU 从事更多的通用计算工作,而不仅仅是 3D 运算





续表 2－1

NT 版本号	操作系统名称	主要特性
NT 6.1	Windows Server 2008 R2	Windows 7 的服务器版本,继续提升了虚拟化、系统管理弹性、网络存取方式以及信息安全等领域的应用,是微软第一个仅支持 64 位的操作系统,支持多达 64 个物理处理器或最多 256 个系统的逻辑处理器
NT 6.2	Windows 8	第一款带有 Metro 界面的桌面操作系统,支持来自 Intel、AMD 和 ARM 的芯片架构
	Windows Server 2012	支持 64 个物理处理器、640 个逻辑处理器、4TB 内存和 64 个故障转移群集节点。包含了一种全新设计的弹性文件系统(Resilient File System,ReFS),是以 NTFS 为基础构建的,支持新一代存储技术,并保持了与 NTFS 的兼容
NT 6.3	Windows 8.1	Windows 8.1 是为 Windows 10 铺路的
	Windows Server 2012 R2	Windows Server 2012 的升级版本,功能性大大增强,支持工作文件夹、状态配置、存储分级、存储定位、回写式高速缓存、重复数据删除技术等新功能
NT 6.4	Windows 10	微软最新的操作系统,支持生物识别等多种技术

这里有必要说明一下,NT 版本号其实是指 Windows 内核的组件版本号。例如 Windows 7 和 Windows Server 2008 R2 虽然分别作为客户机版本和服务器版本,但都是基于 NT 6.1 版本的内核的,它们的 ntoskrnl.exe、hal.dll、设备驱动、协议栈驱动以及相关辅助工具等都是一样的。所不同的是某些默认配置以及针对各自细分应用场景的优化,例如系统劳务线程数量、内存池大小等。

2.2 Windows 操作系统架构

Windows 操作系统按照层次化的思想进行了分层,整合了应用软件、系统软件/服务、设备驱动、硬件隔离层等多方面要素,其整体框架如图 2－3 所示。

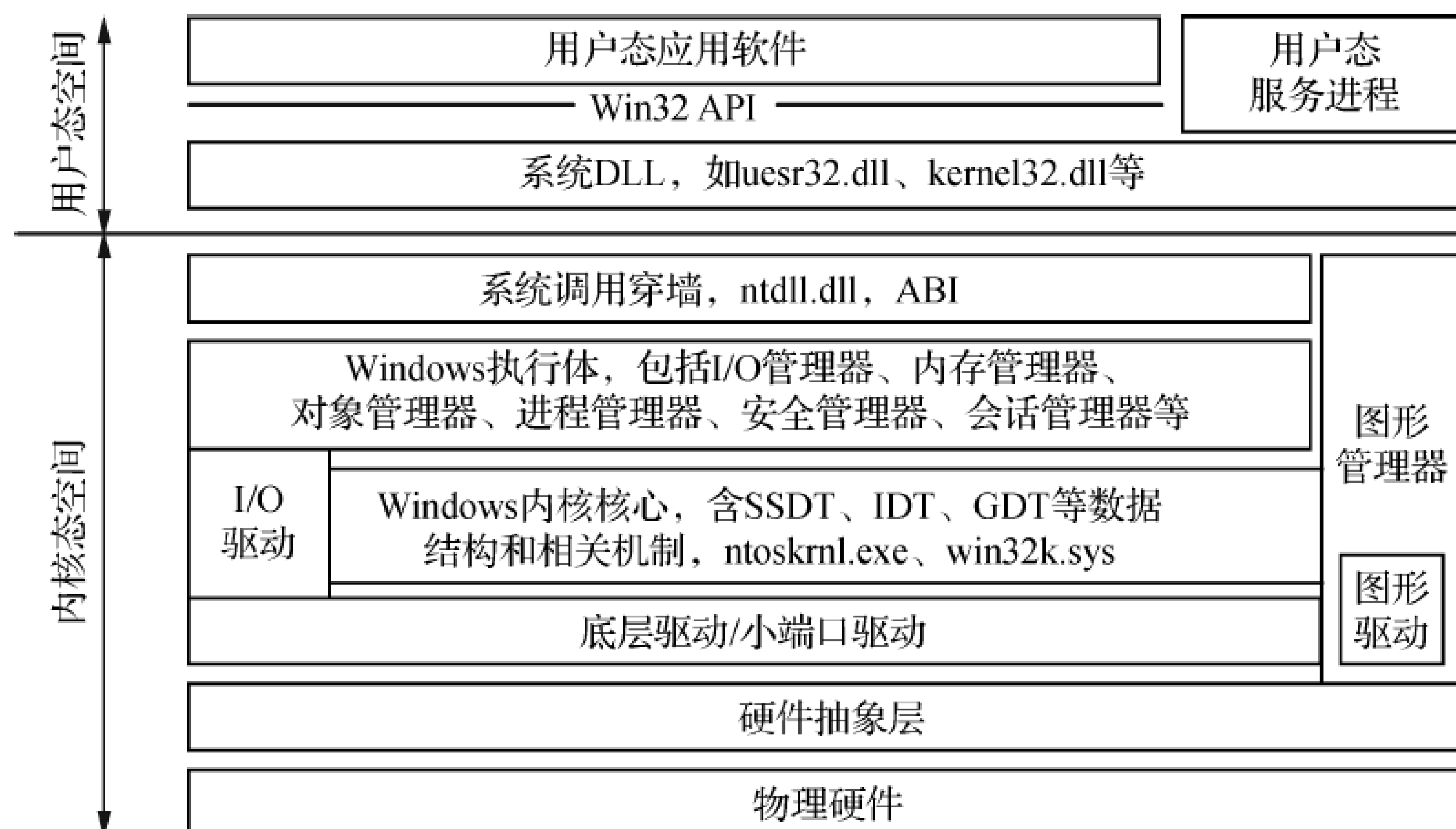


图 2－3　Windows 操作系统整体框架

在 Windows 中虚拟地址空间分为用户态空间(对应 CPU 的优先级是 R3 级别)和内核态空间(也叫系统态空间,对应 CPU 的优先级是 R0 级别)两部分,即在 32 位系统中 0x80000000 以下为用户态空间,0x80000000 以上为内核态空间。Linux 的划分也是如此。当然,这个 0x80000000 的分界点也不是一成不变的,可以通过系统配置进行调整,例如将内核态的低址方向的 1 GB 空间划到用户态,使其空间比为 3∶1。在这里需要注意的是,我们所说的地址空间是内存的虚拟地址空间而非物理地址空间,32 位系统中虚拟地址空间只有 4 GB。关于内存空间的细节后续章节会讲到。而在开启 PAE(物理地址扩展)功能时,操作系统能够管理的物理地址空间可以大于 4 GB,虚拟地址空间仍是 4 GB。更多的物理内存意味着更少的磁盘页面倒换,也意味着更快的访问速度。在 64 位操作系统中,虽然地址线为 64 位,但其实只有 48 位可用,因为 48 位的地址空间(256 TB)已经足够大了,足以支撑目前各种软件的虚拟地址空间。在 64 位系统中,起始地址(0x0)—0x0000 FFFF FFFF FFFF 是用户态空间,0xFFFF 0000 0000 0000—结束地址是内核态空间,中间的大片"飞地"用于隔离这两个空间。

对于 X86 CPU 而言,其优先级在用户态为 Ring3,在内核态为 Ring0。内核态的权限级别很高,因此用户态访问内核态必须通过系统调用或者中断异常、自陷等方式进行,而内核态调用用户态则容易很多,这样设计也是为了提升操作系统的安全性。用户态进程发生了异常最多就是报错并结束该进程,而内核态发生错误或异常则会发生 BSOD(死亡蓝屏),系统必须重启。

简单来说,我们日常开发和使用的应用软件基本都是用户态软件,其 EXE 和依赖的 DLL(动态链接库)等均位于用户态空间,包括 Windows 提供的 user32. dll、kernel32. dll、gdi32. dll、advapi32. dll 等,提供了 Win32 API 接口(Native API),例如 ReadFile 等。但当这些软件需要进行文件读写、内存分配等操作的时候,就需要借助内核态软件的力量了,这个力量被比喻为"凿墙穿透",而使用的"凿子"就是系统调用,例如 ZwReadFile。这些系统调用的入口都在 ntdll. dll 中。ntdll. dll 是个很特殊的 DLL,用户态与内核态的切换都是在这个库中实现的。具体地说,这个库会调用相应的自陷指令(int 0x2E)或者快速调用指令(Intel CPU 下为 sysenter,AMD CPU 下为 syscall),将 SSDT(系统服务描述符表)中的调用号(对应 SSDT 中的具体系统调用,例如 NtReadFile 在表中的索引)赋值到 EAX 寄存器,并进行参数压栈、内核态切换、指令调用等操作,从而进入系统空间。我们可以将 ntdll. dll 看成用户态空间与内核态空间的分水岭,图 2-3 中两种状态的分界线严格来说应该处在 ntdll. dll 模块的中间。

如图 2-4 所示就是在 Win7 64 位系统下 ntdll. dll 导出的接口,可以看到其 Entry Point 都位于用户空间,其接口名称之前都加了 Zw 前缀,以区别于上层 DLL 导出的原生接口。

此处我们要简单介绍下 ABI(Application Binary Interface)。对于 API 大家都很熟悉,而 ABI 描述的是应用程序的二进制接口,更为底层,同时包括了诸如数据类型、对齐规则、系统调用约定、寄存器使用、参数的传递顺序、参数在堆栈中如何存放、系统调用怎样返回等规则,因此 ABI 与实现语言、编译器、操作系统类型、处理器类型等都有关系。ABI 允许编译好

Ordinal ^	Hint	Function	Entry Point
1701 (0x06A5)	1692 (0x069C)	ZwQueueApcThreadEx	0x000215E4
1702 (0x06A6)	1693 (0x069D)	ZwRaiseException	0x000215FC
1703 (0x06A7)	1694 (0x069E)	ZwRaiseHardError	0x00021614
1704 (0x06A8)	1695 (0x069F)	ZwReadFile	0x0001F900
1705 (0x06A9)	1696 (0x06A0)	ZwReadFileScatter	0x0001FCF4
1706 (0x06AA)	1697 (0x06A1)	ZwReadOnlyEnlistment	0x0002162C
1707 (0x06AB)	1698 (0x06A2)	ZwReadRequestData	0x000200AC
1708 (0x06AC)	1699 (0x06A3)	ZwReadVirtualMemory	0x0001FEA0
1709 (0x06AD)	1700 (0x06A4)	ZwRecoverEnlistment	0x00021644
1710 (0x06AE)	1701 (0x06A5)	ZwRecoverResourceManager	0x0002165C
1711 (0x06AF)	1702 (0x06A6)	ZwRecoverTransactionManager	0x00021674
1712 (0x06B0)	1703 (0x06A7)	ZwRegisterProtocolAddressInformation	0x0002168C
1713 (0x06B1)	1704 (0x06A8)	ZwRegisterThreadTerminatePort	0x000216A4
1714 (0x06B2)	1705 (0x06A9)	ZwReleaseKeyedEvent	0x000216C0

图 2－4　ntdll. dll 导出接口示例

的目标代码在使用兼容 ABI 的系统中无需改动就能运行，但 ABI 是汇编级别的接口，不能被高层语言直接拿来用。

在用户态空间，还有一些微软或者第三方提供的服务进程或服务支撑进程，例如 csrss. exe、lsass. exe、smss. exe，以及一些微软合作方提供的诸如显卡加速驱动、网卡加速驱动、文件系统等程序。服务支撑进程并不由服务控制管理器启动，比如登录管理、会话管理，出于性能和安全性的考量，也有许多放在用户态，它们大多符合 Windows UMDF 框架。在具体介绍各服务进程之前先来看看环境子系统的概念。Windows 中的环境子系统包括了 Windows 环境子系统、POSIX（Portable Operating System Interface of UNIX，UNIX 可移植操作系统接口）子系统和 OS/2（IBM 开发的操作系统）子系统三种。早期三种都是存在的，然而随着时代的发展，后两种基本都淡出了 Windows 系统的支持序列，只剩下 Windows 环境子系统一家独大。而 Windows 环境子系统又包括了环境子系统承载进程 csrss. exe、内核设备驱动 win32k. sys、一些子系统 DLL（如 kernel32. dll、advapi32. dll、user32. dll 和 gdi32. dll 等）和一些设备驱动（如图像设备驱动、打印机驱动和视频小端口驱动等）。以下是各个服务进程的介绍：

- **csrss. exe**：csrss. exe 是 Windows 环境子系统的承载进程，是 Windows 子系统在用户态的部分（在内核态的部分是 win32k. sys，这部分负责窗口和图形管理以及对 DirectX 功能的封装），csrss 的全称为 Client/Server Runtime Server Subsystem（客户端/服务端运行时子系统），作用是将执行体的一些功能以子集的方式提供给其他进程调用，并记录所有 Win32 程序的进程和线程的创建与删除事件，也管理与图形相关的任务，同时执行 16 位的虚拟 MS-DOS 环境的图形窗口进程。
- **win32k. sys**：包括窗口管理器（窗口显示、屏幕输出、键盘鼠标输入、向用户传递消息）和图形设备接口（Graphics Device Interface，GDI，图形输出函数库，位于应用程序和图像设备驱动之间，负责翻译应用程序的图像输出请求并将请求发送到图像显示设备，在屏幕上绘制图像等）两部分。
- **lsass. exe**：Windows 安全机制，用于本地安全性授权，比如登录操作，为 winlogon 服务的用户验证生成一个进程等。
- **explorer. exe**：Windows 资源管理器，用于操作系统的图形外壳，包括开始菜单、任务

栏、桌面和文件管理等。

- **services. exe**：用于管理启动和停止服务，该进程也会处理计算机在启动和关闭时运行的服务，并且众多服务进程的父进程都是 services. exe。
- **smss. exe**：smss 的全称是 Session Manager Subsystem，会话管理子系统，是 ntoskrnl（内核）创建的第一个用户态进程，负责启动用户会话，运行 Windows 登录过程，同时启动 Windows 子系统进程 csrss. exe 和登录进程 winlogon. exe（在系统的第一个会话即会话 0 时为 wininit. exe），创建完成后 smss. exe 退出。
- **svchost. exe**：用于加载操作系统提供的. dll 文件，可以运行多个实例（一个服务对应一个实例）。一个 svchost. exe 进程会加载很多. dll 文件，它们可以承载很多功能，例如 RPC（远程过程调用）等。这些动态链接库以 svchost. exe 为宿主可执行文件，向外界进程提供服务，但 svchost. exe 本身不提供任何服务。当系统启动时，svchost. exe 会检查注册表以明确自己需要加载的动态链接库。
- **System Idle Process**：这个特殊进程的进程 ID 永远为 0，称为空闲进程，但没有实际的映像名予以对应（即找不到 idle. exe 这个文件）。这个进程是个单线程进程，负责在每一个处理器上占用 CPU 空闲时间。
- **System Process**：这个特殊进程的进程 ID 永远为 4，称为系统进程，其所属的线程称为内核模式系统线程。这些系统线程就是内核劳务线程/内核工作线程，用于内存平衡集管理、内存脏页面换出等。从图 2－5 中也可以看出，系统线程的发起者和调用的模块一般都是 ntoskrnl. exe（系统内核）和. sys（设备驱动）等内核模块。

System:4 属性

映像 | 性能 | 性能图表 | 磁盘和网络 | GPU图表
线程 | TCP/IP | 安全 | 环境 | 字符串

计数：150

TID	CPU	CSwi...	开始地址
11892			ndis. sys!NetDmaAllocateChannel+0x440
11832			ndis. sys!NetDmaAllocateChannel+0x440
10668			rdbss. sys!RxReference+0x164
10420	< 0. 01	1	rdbss. sys!RxReference+0x164
10116			ntoskrnl. exe!KeIsAttachedProcess+0x1c
9972			ntoskrnl. exe!KeIsAttachedProcess+0x1c
9620			ntoskrnl. exe!KeIsAttachedProcess+0x1c
9524			ntoskrnl. exe!KeIsAttachedProcess+0x1c
8200	< 0. 01	1	ntoskrnl. exe!FsRtlAddToTunnelCache+0x2678
8164			rdbss. sys!RxReference+0x164
7732			ndis. sys!NetDmaAllocateChannel+0x440
5756			ntoskrnl. exe!FsRtlAddToTunnelCache+0x2678
5540	< 0. 01	4	360AvFlt. sys+0x9a60
3996			rdbss. sys!RxReference+0x164
3716			rdbss. sys!RxReference+0x164
2888			HTTP. sys+0x50010
2652			rdbss. sys!RxReference+0x164
2552			srv. sys+0x46910
2544			srv. sys+0x46910

图 2－5　本机系统线程示意图

这里还要解释一下内核劳务线程。内核劳务线程是专门为系统做一些不那么紧急但重复性比较高的专职工作的线程。例如中断处理例程后半段的执行都是放在内核劳务线程中执行的（DPC，延迟过程调用），因为这些工作不应该在高中断优先级环境下运行，而应该在普通优先级状态下执行，因此这些劳务线程也都处于这种较低的优先级状态。系统具备一个工作项（WorkItem）队列，WorkItem 包含一个例程以及一个参数，当内核劳务线程执行时从

队列中获取 WorkItem 并执行这些例程,这个过程比 DPC 本身简单。

内核劳务线程按照线程优先级(区别于中断请求优先级)可分为三类:

➢ **延迟型劳务线程**:线程优先级为 12,处理非紧急工作项,处于等待状态时允许线程的栈内存页面被换出到倒换页面文件中。
➢ **紧急型劳务线程**:线程优先级为 13,处理一些紧急工作项,线程的内存页面不允许倒换。
➢ **超紧急型劳务线程**:线程优先级为 15,处理超紧急工作项,线程的内存页面不允许倒换。

可以通过 ExQueueWorkItem 方法或 IoQueueWorkItem 方法把 WorkItem 分发到一个队列中,也可以通过 ExpWorkerThreadBalanceManager 方法确定是否需要创建新的紧急型劳务线程。

在内核态空间中,最上面一层是 Windows 执行体,包括内存管理器、I/O 管理器、缓存管理器、对象管理器、PNP(Plug-and-Play,即插即用)管理器、进程管理器、安全管理等,如图 2－6 所示,它是 ntoskrnl. exe 模块偏上面的一层。例如后文要介绍的完成端口机制,就是 I/O 管理器的组成部分。Windows 执行体中包括了一些管理机制,用于将用户态的服务请求转换成更为底层的服务调用。

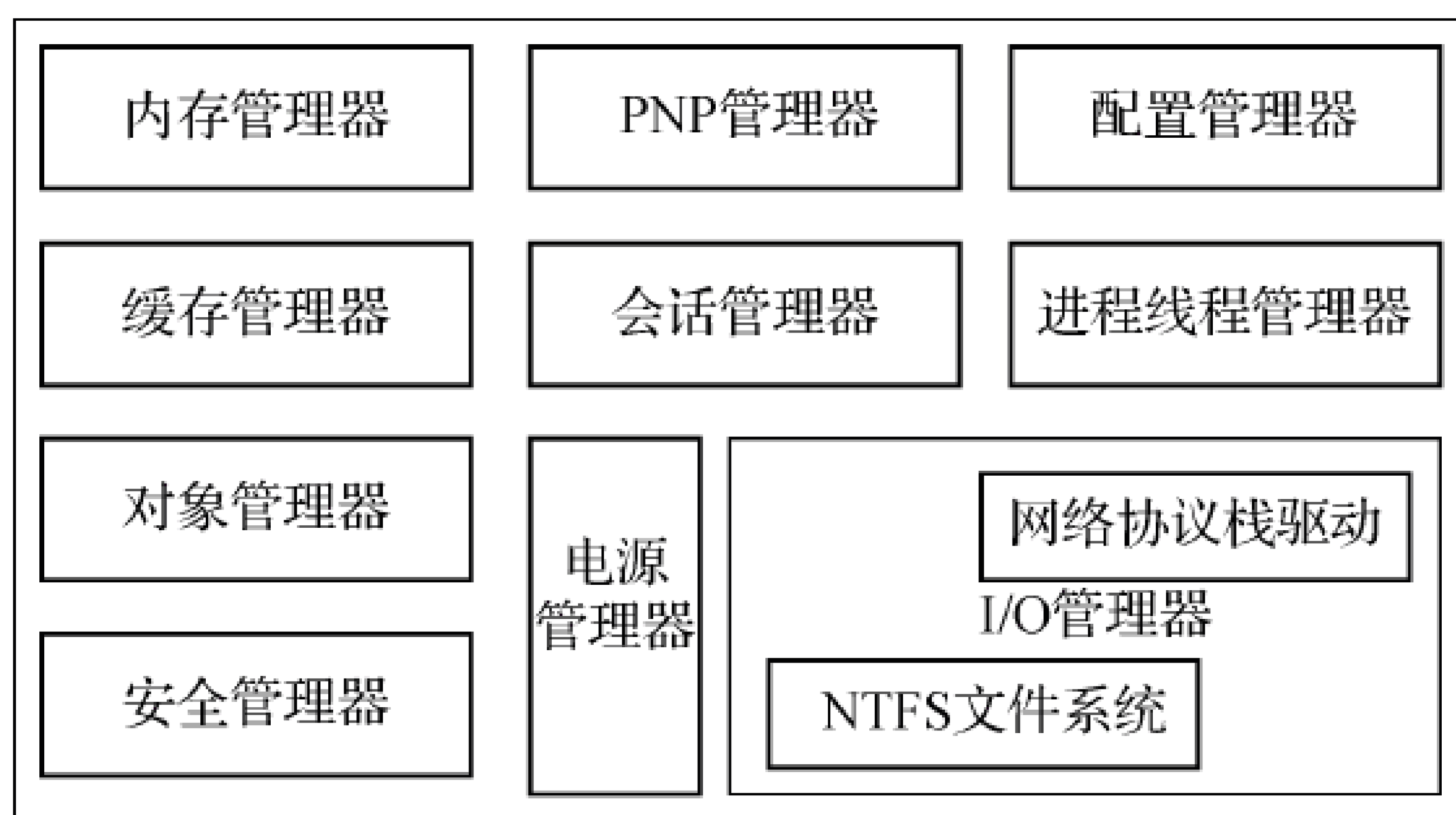


图 2－6　Windows 执行体框架

Windows 执行体包含了下列主要组件:

➢ **配置管理器**:管理系统中的注册表。
➢ **进程线程管理器**:创建和终止进程线程,对于内核核心层的进线程管理则提供了更多的语义支持。
➢ **I/O 管理器**:将 I/O 请求翻译为相应设备/协议栈驱动能够理解的语义(I/O 请求包,IRP),包括网络协议栈、NTFS 文件系统协议栈和相应 I/O 设备的协议栈的上层结构。
➢ **PNP 管理器**:枚举总线上的设备;获取每个设备需要的硬件资源,例如端口(Port)、中断请求(Interrupt Request,IRQ)等,并分配这些资源(由总线驱动向 PNP 管理器报告其总线上的设备);加载相应设备的驱动程序;对于设备的移除和停用,发送通知消息到系统中。

➢ **会话管理器**：本地或者远程登录 Windows 时，系统都会分配一个会话 ID。所谓会话，就是我们登录之后的运行环境，这个环境的标识就是会话 ID，是与每个登录相关联的。会话管理器是系统中创建的第一个用户态进程，它负责准备系统登录所需要的环境、win32k. sys，并启动 csrss. exe 和 winlogon. exe 进程，再由 winlogon. exe 启动其他的系统管理进程，最后进入登录界面。

➢ **电源管理器**：管理电源事件，产生电源 I/O 通知并发送给设备驱动程序。

➢ **内存管理器**：实现了虚拟内存机制，为进程提供虚拟内存空间，管理从虚拟内存向物理内存的映射。

➢ **缓存管理器**：管理磁盘文件与内存文件的互换与映射，提高系统访问磁盘文件的速度。

➢ **对象管理器**：可以提供一种公共的机制来使用系统资源，将散落在系统各个角落中的资源控制操作集中在一起，可以对对象进行计数、限制使用和统一命名等。Windows 内部的对象包括执行体对象和内核对象两种，其中执行体对象就是上文说的对象管理器、进程管理器、I/O 管理器等，而内核对象则是更基础、更底层的一些对象，例如内存对象、文件对象、互斥量对象等，内核对象对于用户态软件是不可见的，只能由执行体文件使用。图 2－7 是导出给用户态软件使用的执行体对象。

Name	Type
Adapter	Type
ALPC Port	Type
Callback	Type
Controller	Type
DebugObject	Type
Desktop	Type
Device	Type
Directory	Type
Driver	Type
EtwConsumer	Type
EtwRegistration	Type
Event	Type
EventPair	Type
File	Type
FilterCommunicationPort	Type
FilterConnectionPort	Type
IoCompletion	Type
IoCompletionReserve	Type
Job	Type
Key	Type
KeyedEvent	Type
Mutant	Type
PcwObject	Type
PowerRequest	Type
Process	Type
Profile	Type
Section	Type
Semaphore	Type
Session	Type
SymbolicLink	Type
Thread	Type
Timer	Type
TmEn	Type
TmRm	Type
TmTm	Type
TmTx	Type
Token	Type
TpWorkerFactory	Type
Type	Type
UserApcReserve	Type
WindowStation	Type
WmiGuid	Type

图 2－7　使用 WinObj 枚举的执行体对象

➢ **安全管理器**：守护操作系统资源，执行对运行时对象的保护和审计。

除此之外，在 Windows XP 系统中，还包括 Prefetch 预取管理器，在 Windows Vista 之后它演进为 Superfetch 管理器。预取技术的基本思路是：在载入某个程序之前，预先从硬盘中载

入一部分该程序运行时所需的数据到物理内存中,这样便能加快程序的启动速度。当然,这样做会拖累内存使用,用户可以选择关闭预取策略。

执行体的下一层就是我们常说的 Windows 内核核心,这一层是 Windows 系统中最重要的部分,包括了许多数据结构、内核对象、执行代码、系统调度和运行机制、异常处理和分发机制、多处理器同步机制等。例如 SSDT、IDT、GDT、TSS 等核心数据结构,DPC 机制,APC 机制,系统调用相关机制,线程切换机制,异常处理相关机制等均在这一层实现,具体的承载者是 ntoskrnl. exe 中偏下面的一层。ntoskrnl. exe 虽然是 EXE 文件,但却不是一般意义上的可执行文件,. exe 后缀名只是表示它符合 Windows PE 文件格式并输出了一些接口而已。而 win32. sys 是 Windows 子系统在内核态的部分,主要负责与图形图像相关的处理。对于进程和线程,在用户态层、执行体层和内核核心层分别有相应的数据结构进行描述,例如线程的 TEB、ETHREAD、KTHREAD 数据结构,分别对应上述三个层次,以方便每个层次的访问。

设备驱动一般分为硬件设备驱动、文件系统驱动、网络协议栈驱动、过滤型驱动等几种类型。而 I/O 驱动主要是指与输入输出相关的驱动程序,其中包括 I/O 设备直接相关的驱动、协议栈驱动、磁盘驱动、文件系统驱动等。文件系统驱动和协议栈驱动一部分在执行体中,与 I/O 管理器关系密切;另一部分是与内核核心层平齐的,例如协议栈驱动的中间部分。而过滤型驱动的主要作用是针对 IRP(包括文件读写和网络读写)进行诸如加解密、增加附加信息、修改数据结构等操作。

内核核心层的下一层是各种小端口驱动和其他底层驱动,例如网卡小端口驱动、输入类设备驱动、磁盘小端口驱动和其他一些小端口/底层驱动。这部分驱动向上与内核核心和 I/O 类驱动打交道,向下与 HAL 打交道,基本上是一些类驱动,作为底层硬件的共性服务驱动。

HAL(Hardware Abstraction Layer,硬件抽象层)是微软为了保证整个系统的兼容性和稳定性而提出的,其承载的库文件是 hal. dll。内核核心层也要依赖 hal. dll,同时 hal. dll 也依赖于内核核心层,如图 2-8 和2-9 所示。HAL 的设计初衷是内核核心不需要直接与底层硬件(对应的硬件驱动)直接打交道,将硬件差别与操作系统进行抽象解耦,因此叫作硬件抽象层,这也是层次化思想在 Windows 中的具体体现。同时,移植一种新的硬件到系统中时,需要安装的驱动模块也可以做到最简化,因为大部分共性服务已经在 HAL 及其上层中实现了,底层设备的厂家只需要按照 HAL 的接口与数据结构进行对接即可。HAL 的下层就是硬件层,主要包括各种型号设备的最底层驱动,实现的是针对具体设备的个性化功能,例如某型网卡的驱动等。

综上所述,Windows 系统是个分层的系统,但是各个层次中也夹杂着一些跨越了几个层次的垂直条带,这是由历史和技术限制共同造成的。例如网络协议栈驱动和 I/O 管理器共同位于执行体层,因为涉及的内核核心部分很少,因此其协议栈垂直地从执行体层跨越到小端口驱动层,甚至直接跨越到 HAL。这样虽然打乱了分层的清晰性,但也没有什么特别不合理的地方。Windows 系统毕竟是一套完善的兼容于各种处理器的庞大操作系统,因此也不

图 2－8　在 Win7 64 位系统中 ntoskrnl. exe 与 hal. dll 的依赖关系

图 2－9　hal. dll 也反向调用 ntoskrnl. exe 中的接口

可能做到全盘化的条带分离。

另外，Windows 也是一个基于对称多处理器（SMP）架构的操作系统，其使用的所有处理器共享统一的主存（但每个处理器都有自己的缓存），且无论内核态线程还是用户态线程均

可以被调度到任何处理器上运行[与非对称多处理器(ASMP)架构的操作系统不同,ASMP架构操作系统的内核态代码需要在主 CPU 上运行,用户态代码只能在其他 CPU 上运行]。同时,Windows 也支持多核、超线程和非统一内存存取(NUMA)三种特性。针对超线程架构,原本 Windows 将 CPU 的物理核心看作线程调度的承载单元,而现在将逻辑核心看作承载单元;对于 NUMA 架构,Windows 仍然将其看作基于对称多处理器的架构形态,用户态和内核态线程仍然可以被调度到任何处理器上运行,只是访问本地主存的速度快一点,访问远端主存的速度稍慢一点而已。

2.3 WoW64

WoW64(Windows-on-Windows 64-bit)是一个 Windows 子系统,为 32 位应用程序提供了模拟执行环境(32 位环境),相当于在 64 位环境下建立了一个 32 位的子系统,是 64 位系统中 Win32 系统的仿真,以方便大多数应用程序无需修改而直接运行在 Windows 64 位系统上。WoW64 是用户态空间的模块,位于 32 位 ntdll. dll 之下、内核执行体层之上,由三个 DLL 支持实现:

- **wow64. dll**:负责 32 位进程线程创建,并且挂钩异常分发函数和 64 位 ntdll. dll、ntoskrnl. exe 导出的基本系统调用,同时也支持文件和注册表重定向。
- **wow64win. dll**:挂钩了 win32k. sys、win32u. dll 导出的 GUI 系统调用。
- **wow64cpu. dll**:负责 CPU 的 32 位和 64 位的切换,提供与 CPU 体系结构相关的支持。

WoW64 在 64 位系统中的位置如图 2-10 所示,其所有库以及模拟出来的 32 位运行环境都位于用户态内存空间。

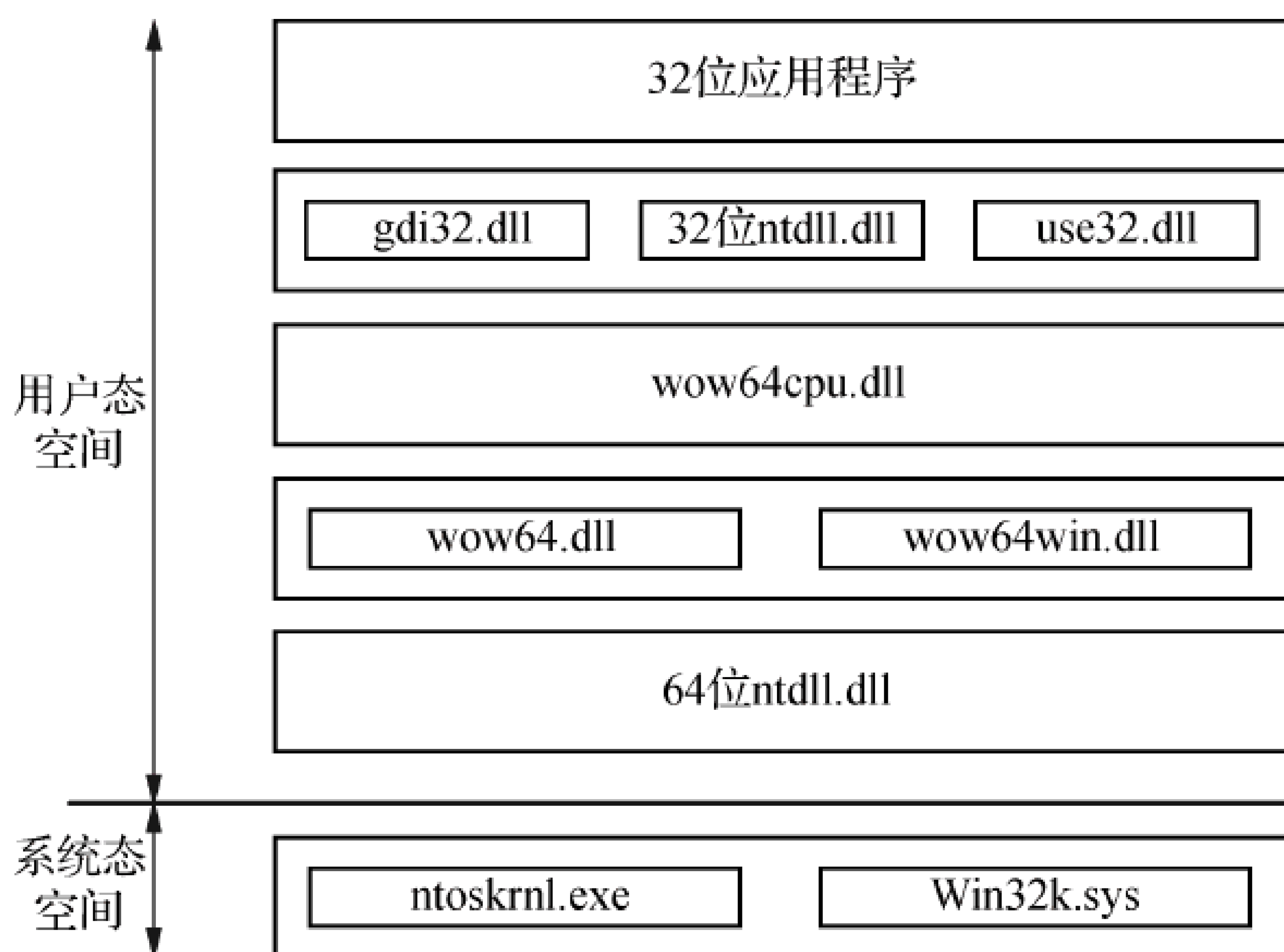


图 2-10　Wow64 在 64 位 Windows 系统中的位置

在进程创建过程中,进程管理器将原生的 64 位的 ntdll. dll 和 32 位的 ntdll. dll 一起映射到虚拟地址空间中。进程初始化的时候,会首先调用 wow64. dll 的进程初始化代码,然后建

立 32 位的运行环境(例如 CPU 切换到 32 位模式下,开始执行 32 位的 DLL 加载器),之后的初始化和运行过程与 32 位原生系统的一模一样。

同时,64 位原生系统库位于 64 位 Windows 系统的 Windows\System32 目录下,也同样需要将该目录映射到 Windows\Syswow64 目录中,那里面存放有 32 位用户态的系统库。例如在作者的系统中以 OllyDbg 运行某应用程序,可以看到在可执行模块加载序列中,除了在 EXE 根目录下的业务库外,还加载了 Windows\Syswow64 目录下 32 位的系统库和 Windows\System32 目录下 64 位的原生系统库,如图 2 - 11 所示。中间互通的桥梁就是 WoW64。

虽然其他的 32 位系统库仍然是调用 32 位的 ntdll. dll(Windows\Syswow64 目录下),但在 32 位的 ntdll. dll 进入系统调用时就不直接调用 sysenter(后文详细介绍)命令了,而变成了 call TEB. WOW32Reserved,这个字段保存了 32 位向 64 位切换的 Shell Code(跳板)地址。这个跳板中包含一个长跳转,以跳转到 wow64cpu. dll 中,这是个 64 位的模块。在完成了 CPU 从 32 位向 64 位模式切换以后便开始调用 64 位的原生 ntdll. dll 了。

Base (系统)	大小 (十进制)	条目	名称	类型	文件版本	静态链接	路径
00220000	0001D000 (118784.)	0022969E	ZxLog			KERNEL32,	E:\平台版本\测试环境4.0.005服务器(GB28181
00250000 (系统	00038000 (229376.)		odbcint		6.1.7600.16385		C:\Windows\system32\odbcint.dll
00330000	000C6000 (811008.)	003A1E8D	log4cxx_znv			ADVAPI32,	E:\平台版本\测试环境4.0.005服务器(GB28181)\log4
00400000 (main)	00039000 (233472.)	00418C69	cmsserver		1, 2, 0, 0	ACE_znv, C	E:\平台版本\测试环境4.0.005服务器(GB28181)\cm
00440000	000E3000 (929792.)	004C1A51	TCPModule		1, 2, 0, 1	ACE_znv, C	E:\平台版本\测试环境4.0.005服务器(GB28181)\TC
00640000	00142000 (1318912.)	006FBCE1	ACE_znv		5.5	ADVAPI32,	E:\平台版本\测试环境4.0.005服务器(GB28181)\
00790000	0007B000 (503808.)	007D142A	CommonModule		1, 2, 0, 0	ACE_znv, K	E:\平台版本\测试环境4.0.005服务器(GB28181)\Commo
10000000	00084000 (540672.)	10042B49	workFactry		1, 2, 0, 0	ACE_znv, C	E:\平台版本\测试环境4.0.005服务器(GB28181)\wor
12000000	00242000 (2367488.)	121195DA	xerces-c_2_7		2, 7, 0	ADVAPI32,	E:\平台版本\测试环境4.0.005服务器(GB28181)\xerce
6EFA0000 (系统	0008C000 (573440.)	6EFD0EC2	ODBC32		6.1.7601.17514	ADVAPI32,	C:\Windows\system32\ODBC32.dll
6F4B0000 (系统	00038000 (229376.)	6F4B990E	fwpuclnt		6.1.7600.16385	KERNEL32,	C:\Windows\System32\fwpuclnt.dll
6F4F0000	00021000 (135168.)	6F4F4F16	mdnsNSP		3,1,0,1	KERNEL32,	C:\Program Files (x86)\Bonjour\mdnsNSP.dll
6F520000 (系统	00012000 (73728.)	6F5218F2	pnrpnsp		6.1.7600.16385	API-MS-Win	C:\Windows\system32\pnrpnsp.dll
6FB60000 (系统	00005000 (20480.)	6FB615DF	wshtcpip		6.1.7600.16385	ntdll, WS2	C:\Windows\System32\wshtcpip.dll
6FB70000 (系统	0003C000 (245760.)	6FB7145D	MSWSOCK		6.1.7600.16385	API-MS-Win	C:\Windows\system32\MSWSOCK.dll
6FE20000 (系统	00006000 (24576.)	6FE214B2	rasadhlp		6.1.7600.16385	API-MS-Win	C:\Windows\system32\rasadhlp.dll
71310000 (系统	00008000 (32768.)	7131131E	winrnr		6.1.7600.16385	API-MS-Win	C:\Windows\System32\winrnr.dll
71320000 (系统	00010000 (65536.)	71321526	napinsp		6.1.7600.16385	API-MS-Win	C:\Windows\system32\napinsp.dll
71330000 (系统	00010000 (65536.)	713338C1	NLAapi		6.1.7601.24000	API-MS-Win	C:\Windows\system32\NLAapi.dll
73430000 (系统	00044000 (278528.)	7344640D	DNSAPI		6.1.7600.16385	API-MS-Win	C:\Windows\system32\DNSAPI.dll
74290000 (系统	0000F000 (61440.)	742912A1	wkscli		6.1.7601.17514	KERNEL32,	C:\Windows\system32\wkscli.dll
742A0000 (系统	00019000 (102400.)	742A1319	srvcli		6.1.7601.17514	API-MS-Win	C:\Windows\system32\srvcli.dll
742C0000 (系统	00009000 (36864.)	742C15A6	netutils		6.1.7601.17514	KERNEL32,	C:\Windows\system32\netutils.dll
742D0000 (系统	00011000 (69632.)	742D1300	NETAPI32		6.1.7601.17887	API-MS-Win	C:\Windows\system32\NETAPI32.dll
74A50000 (系统	00007000 (28672.)	74A5128D	WINNSI		6.1.7601.23889	API-MS-Win	C:\Windows\system32\WINNSI.DLL
74A60000 (系统	0001C000 (114688.)	74A6A431	Iphlpapi		6.1.7600.16385	API-MS-Win	C:\Windows\system32\Iphlpapi.DLL
74B70000	???		Mod_74B7				
74B80000	???		Mod_74B8				
74BE0000	???		Mod_74BE				
74C40000	0000C000 (49152.)	74C410E1	CRYPTBASE		6.1.7601.24308	API-MS-Win	C:\Windows\syswow64\CRYPTBASE.dll
74C50000	00060000 (393216.)	74C6A3B0	SspiCli		6.1.7601.24308	API-MS-Win	C:\Windows\syswow64\SspiCli.dll
74D00000	00100000 (1048576.)	74D1B6FA	USER32		6.1.7601.17514	ADVAPI32,	C:\Windows\syswow64\USER32.dll
75240000	0015F000 (1437696.)	7528B955	ole32		6.1.7601.24291	API-MS-Win	C:\Windows\syswow64\ole32.dll
757B0000	000F0000 (983040.)	757C0569	RPCRT4		6.1.7600.16385	API-MS-Win	C:\Windows\syswow64\RPCRT4.dll
758A0000	00090000 (589824.)	758B633B	GDI32		6.1.7601.24308	API-MS-Win	C:\Windows\syswow64\GDI32.dll
75930000	00047000 (290816.)	759374E1	KERNELBASE		6.1.7601.18015	ntdll	C:\Windows\syswow64\KERNELBASE.dll
75980000	00035000 (217088.)	7598145D	WS2_32		6.1.7600.16385	API-MS-Win	C:\Windows\syswow64\WS2_32.dll
75A10000	00091000 (593920.)	75A1406E	OLEAUT32		6.1.7601.24117	API-MS-Win	C:\Windows\syswow64\OLEAUT32.dll
75B40000	000AC000 (704512.)	75B4A472	msvcrt		7.0.7601.17744	API-MS-Win	C:\Windows\syswow64\msvcrt.dll
75C70000	00110000 (1114112.)	75C83346	kernel32		6.1.7601.18015	API-MS-Win	C:\Windows\syswow64\kernel32.dll
75D90000	0000A000 (40960.)	75D936A0	LPK		6.1.7601.24280	API-MS-Win	C:\Windows\syswow64\LPK.dll
75DB0000	000CD000 (839680.)	75DB168B	MSCTF		6.1.7600.16385	API-MS-Win	C:\Windows\syswow64\MSCTF.dll
75F40000	00019000 (102400.)	75F44975	sechost		6.1.7600.16385	API-MS-Win	C:\Windows\SysWOW64\sechost.dll
75F70000	000A1000 (659456.)	75F84919	ADVAPI32		6.1.7601.24308	API-MS-Win	C:\Windows\syswow64\ADVAPI32.dll
76020000	00006000 (24576.)	76021782	NSI		6.1.7601.23889	API-MS-Win	C:\Windows\syswow64\NSI.dll
762F0000	0009D000 (643072.)	7632474C	USP10		1.0626.7601.238	GDI32, KER	C:\Windows\syswow64\USP10.dll
767A0000 (系统	00060000 (393216.)	767B158F	IMM32		6.1.7601.17514	API-MS-Win	C:\Windows\system32\IMM32.DLL
77800000	???		Mod_7780	隐			
779C0000	00180000 (1572864.)		ntdll		6.1.7600.16385		C:\Windows\SysWOW64\ntdll.dll
77BF0000 (系统	00066000 (417792.)	77BF1249	MSVCP60		7.0.7600.16385	API-MS-Win	C:\Windows\system32\MSVCP60.dll

图 2 - 11　某 32 位应用软件加载后的内存模块列表

WoW64 是通过挂钩技术实现从 32 位到 64 位切换的,包括系统调用、APC 分发、异常分发等。由于 WoW64 对相关的库进行了挂钩,并且需要从 32 位切换到 64 位,例如堆栈处理、参数处理、调用框架切换等,因此多了很多切换的调用步骤,会一定程度上影响执行效率。WoW64 进程包含 2 个版本的 ntdll. dll。第一个版本是 32 位的,负责将系统调用转到 WoW64 环境下,并在该环境下调整调用框架以适应 64 位的 ABI;第二个版本就是 64 位系统的原生 ntdll. dll 了,它是被 WoW64 环境调用的,负责用户态模式到内核态模式的切换,WoW64 的运行视图如图 2 - 12 所示。本文涉及的挂钩技术在后面章节会有详细描述。

在 WoW64 体系结构下，进程空间、系统调用等机制的实现都是与32位不同的：

- **WoW64 进程的地址空间布局**：WoW64 进程的地址空间可以是 2 GB，也可以是 4 GB，这取决于是否开启大地址空间感知标志。
- **WoW64 系统调用**：可执行文件首先加载 64 位的 ntdll. dll，再加载 32 位的 ntdll. dll，然后加载模式转换的 DLL（wow64. dll、wow64win. dll 和 wow64cpu. dll）和可执行文件本身依赖的诸多模块。线程使用 wow64cpu！X86SwitchTo64BitMode 函数从 32 位切换到 64 位，返回时则通过 wow64cpu！CpupReturnFromSimulatedCode 函数从 64 位切换回 32 位。
- **文件系统重定向**：为了降低应用程序移植代价，所有相关 API 将 Windows\System32 目录替换为 Windows\Syswow64，使用目录重定向实现文件系统重定向。

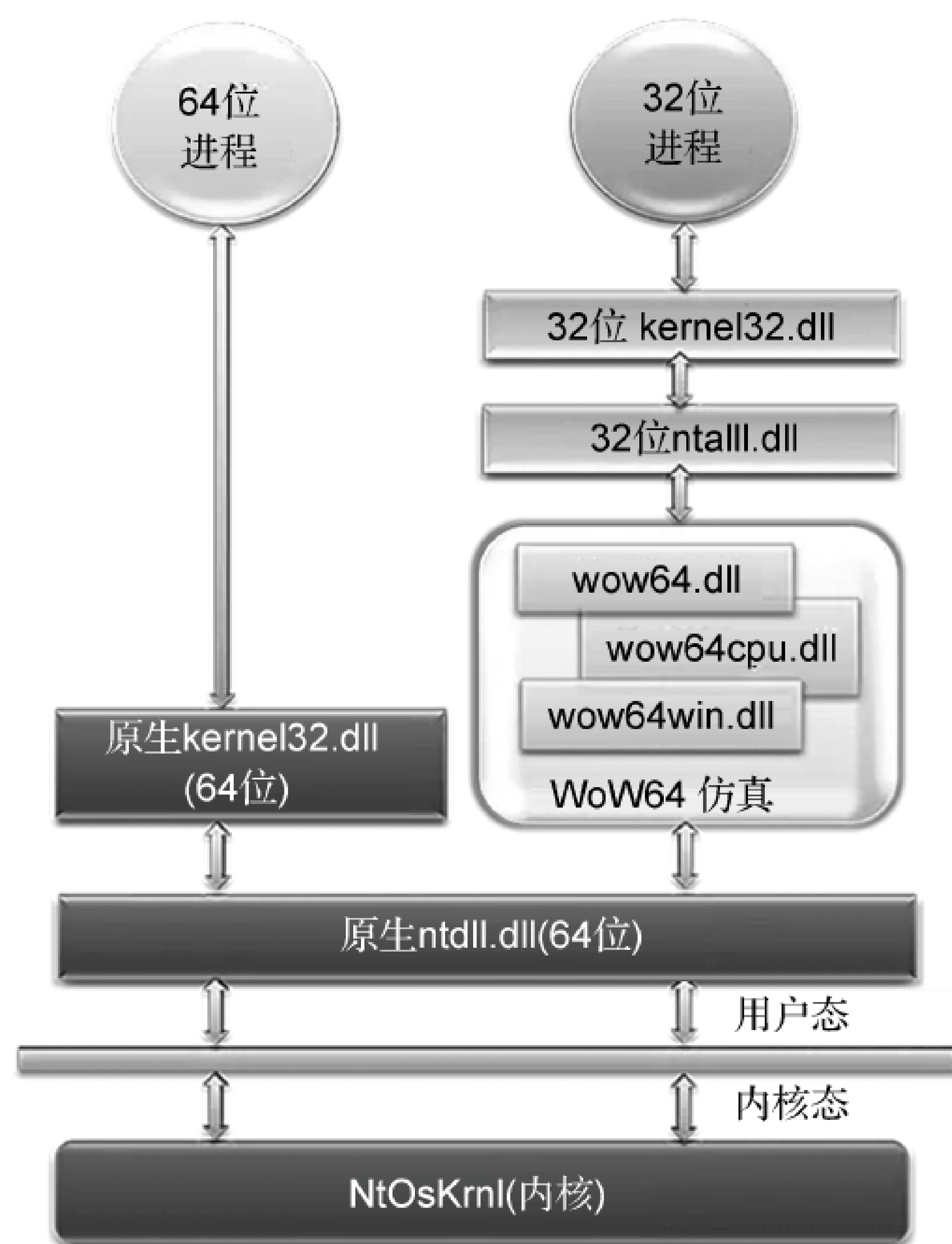


图 2-12　32 位进程在 64 位环境下的运行视图（图片来自 CSDN）

本章小结

Windows 是目前应用最为广泛的系统，具有悠久的历史，在桌面机和服务器中均有广泛的布局。特别是从 NT5.0 内核开始的 Windows 2000 及其后续版本的系统在当前的 X86/X64 平台中占有决定性的份额，其与 Intel 的组合也被称为“Wintel 联盟”。

本章首先简要介绍了 Windows 系统的历史沿革，然后讲述了系统架构，包括两个空间（用户态空间和内核态空间）、各种管理器以及各种系统服务，最后介绍了作为 32 位和 64 位进程转换媒介的 WoW64 系统的架构和功能。

第 3 章　Windows 系统调用

系统调用是 Windows 操作系统中最为重要的内容,其直接描述了用户态进程/线程使用系统资源的机制、规则和框架,包括用户态(R3)和内核态(R0)的切换机制、堆栈切换机制、栈帧的形成与消亡、内核中各种数据结构的配合、调用序列等。本章我们将以 Win32 API 的 ReadFile 接口为例,按照图 3-1 所示提纲进行详细介绍。



图 3-1　本章提纲

3.1　预备知识

本章中我们先介绍一些基础知识,包括寄存器、堆栈、调用约定以及 Windows 系统中重要且常用的数据结构。介绍的时候也会视情况将这些数据结构的使用场景加以说明,以便加深读者对这些基础数据结构、调用机制和寄存器的理解。

3.1.1 寄存器

在 X86 架构下,寄存器大致分为以下几类:

➢ **数据寄存器**:都是 32 位的,但也可以作为 16 位寄存器或者 8 位寄存器使用。

- **EAX**:一般作为累加器使用,也常作为函数返回值的承载寄存器,保存 32 位地址等数据内容也没有什么问题,并没有什么特别的使用局限。
- **EBX**:一般作为地址指针寄存器,也可以用来存储数据内容。
- **ECX**:常用作计数器,特别是在执行循环的时候用于保存循环次数。
- **EDX**:既可以用来保存数据,也可以用来保存地址,没有使用局限。

➢ **变址寄存器**:常用于字符串或者数据块的拷贝移动。

- **ESI**:常指向被移动/拷贝的字符串(Source)。
- **EDI**:常指向字符串拷贝的目的地址(Destination)。

➢ **堆栈指针寄存器**:这是两个比较重要的寄存器,一般用来进行堆栈操作而不能被用于其他操作。

- **ESP**:指向堆栈栈顶,即堆栈当前弹入、弹出的位置。
- **EBP**:扩展基址指针寄存器,指向某个栈帧的基址。一个堆栈中可能有多个栈帧,EBP 指向最靠近栈顶的栈帧的基址(当前栈帧的低址)。

➢ **标志寄存器 EFLAGS**:用于控制任务状态、模式切换、中断处理、指令追踪和访问权限管理,寄存器中的标志位需要特权指令代码才可以修改。EFLAGS 寄存器各位的含义如图 3-2 所示。

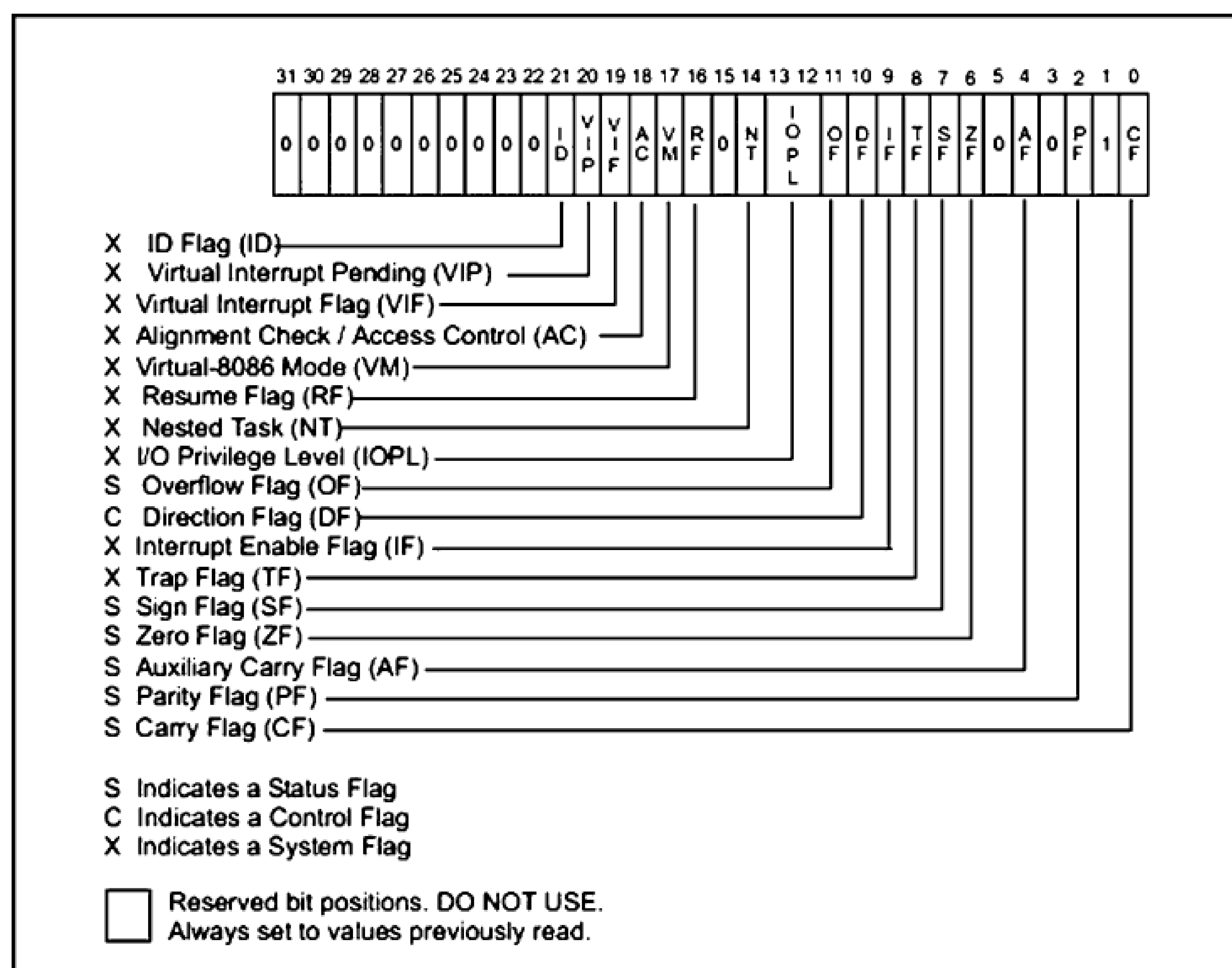


图 3-2 标志寄存器 EFLAGS 示意图(图片来自 CSDN)

- **段寄存器**:内存逻辑地址是由段寄存器的值(内含段基址)加上一个偏移量(Offset)组成的,以便使用较少的位数表示一个很大的地址空间,这种表示方式正是根据内存分段的管理模式而设置的。段寄存器也是比较特殊的寄存器,为了加快段的访问速度,段寄存器分为了两部分。一部分是 16 位的可见部分,如下面的 6 个段寄存器所示,表示某个段在 GDT/LDT 中的段选择子(Selector),其中会有一位表示选择 GDT 还是 LDT;另一部分是 64 位的不可见部分,是段寄存器指向的 Selector 在 GDT/LDT 中具体段描述符的值,之所以要把具体段描述符的值存于此就是为了加快访问速度。由于每个 GDT/LDT 描述符是 8 个字节,因此不可见部分也占用 8 个字节(64 位)。
  - **CS**:即 Code Segment Register,16 位代码段寄存器,保存了当前线程的用户态/内核态空间代码段的选择子。
  - **DS**:即 Data Segment Register,16 位数据段寄存器,保存了当前线程的数据段(静态数据或者全局数据段)的选择子。
  - **ES**:即 Extra Segment Register,16 位附加段寄存器,保存了当前线程的附加数据段的选择子。
  - **FS**:即 Flag Segment Register,16 位标志段寄存器。在用户态下,FS 段值是当前线程的线程环境块(TEB)在 GDT 中的选择子;在内核态下,FS 段值是系统的内核处理器控制区(KPCR)在 GDT 中的选择子,KPCR 区域中保存了与处理器相关的重要域值,如 GDT 与 IDT 的线性地址等,后文将有详细描述。
  - **GS**:即 Global Segment Register,16 位全局段寄存器。
  - **SS**:即 Stack Segment Register,16 位堆栈段寄存器,保存了当前线程的堆栈段选择子。
- **控制寄存器**:决定处理器的操作模式和当前执行任务的一些特征。
  - **CR0**:保存控制系统的工作模式和处理器的状态。
  - **CR1**:保留未用。
  - **CR2**:保存异常出错的线性地址。
  - **CR3**:保存当前进程页目录基址的物理地址和 PCD/PWT 标志位(与 Cache 有关)。
  - **CR4**:保存一些结构的扩展,表明对于特定的处理器和操作系统的执行支持情况。
- **调试寄存器**:主要作用是调试应用程序代码、系统程序代码,开发多任务操作系统,并监视代码的运行和处理器的性能。包含以下 8 个寄存器:DR0、DR1、DR2、DR3、DR4、DR5、DR6、DR7,这里不做详细描述。
- **系统地址寄存器**:
  - **GDTR**:全局描述符表寄存器,48 位,用来存放全局描述符表(GDT)的 32 位线性基地址和 16 位的界限值(limit)。
  - **IDTR**:中断描述符表寄存器,48 位,用来存放中断描述符表(IDT)的 32 位线性基地址和 16 位的界限值。

- **LDTR**:局部描述符表寄存器,16 位,用来存放局部描述符表(LDT)的 16 位选择子。LDTR 还附有一个隐含的描述符高速缓冲寄存器(64 位),用来存放 LDT 中所选段描述符(8 字节)的值,目的只是为了加快访问速度。
- **TR**:任务状态段寄存器,16 位,用来存放任务状态段(TSS)的 16 位选择子。相应地,它也有一个隐含的描述符高速缓冲寄存器(64 位),用来存放 TSS 的段描述符(8 字节)。

CPU 在写入上述两个寄存器时,也填充了隐藏的部分。系统地址寄存器示意图如图 3 - 3 所示。

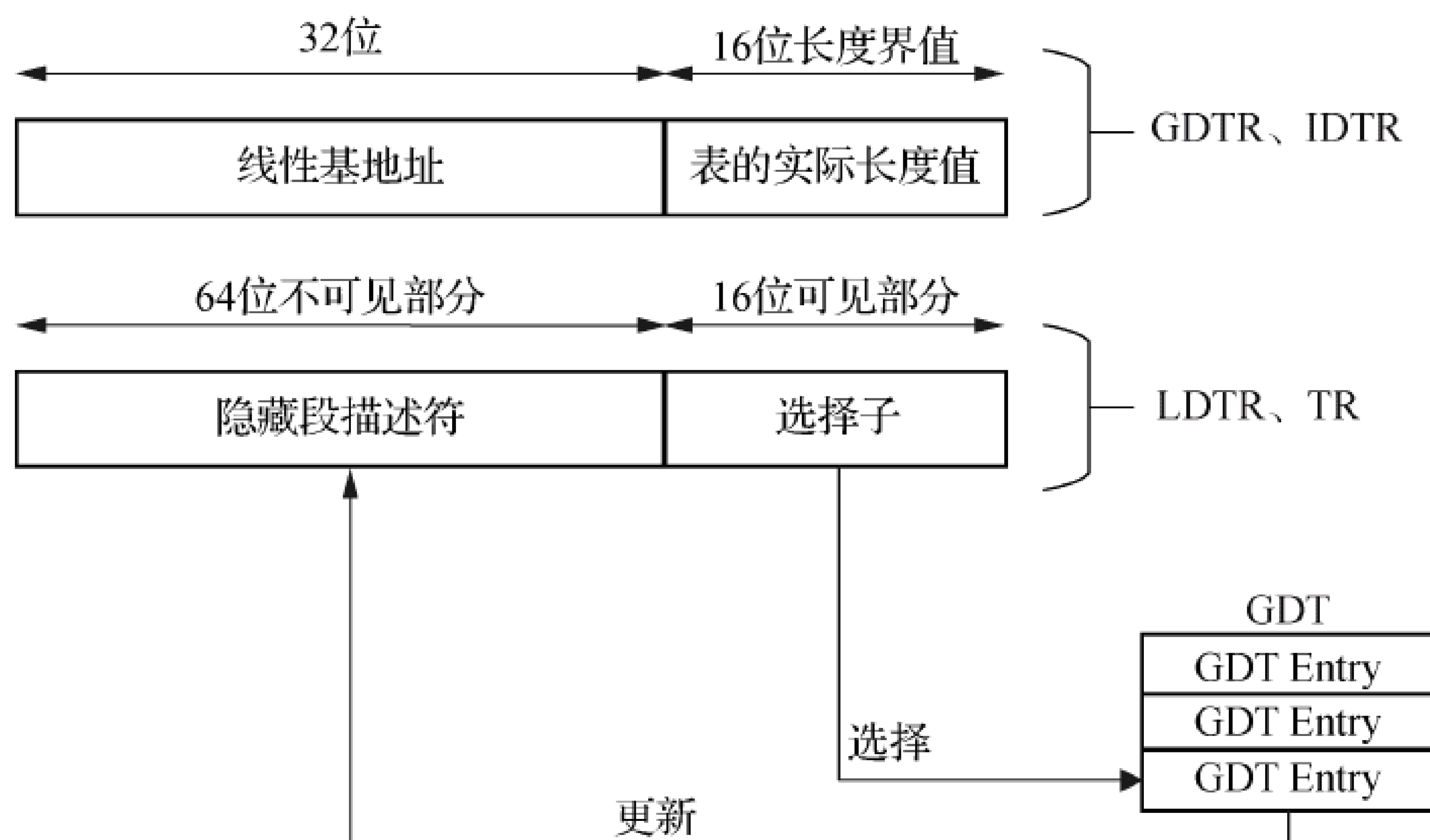


图 3 - 3　系统地址寄存器示意图

➢ **指令指针寄存器 EIP**:32 位寄存器,低 16 位称为 IP,用于兼容 16 位 CPU,其存储的内容是下一条要读入 CPU 的指令在虚拟内存中的偏移地址。当一个程序开始运行时,系统把 EIP 清零。每读入一条指令,EIP 自动增加所读指令的字节数目。在线程切换、APC(异步过程调用)、挂钩等机制中会频繁使用 EIP 寄存器。

这些寄存器既是为了满足 X86 体系结构的要求,也是为了加快数据和代码访问速度的需要。因为寄存器是嵌入到处理器中的,与 CU(计算单元)的连接是通过高速总线(例如 QPI)实现的,所以比访问主存的速度提高至少一个数量级。而对于寄存器中具体存什么数据,既与处理器厂家的硬性要求有关,也与操作系统定义的灵活性有关。

X64 体系结构下,寄存器也做了相应的扩展和更改,包括:

➢ **寄存器宽度扩展**:除了段寄存器和 EFLAGS 寄存器仍然是 32 位的外,其他寄存器都扩展为 64 位的。

➢ **寄存器数量扩展**:新增了 8 个通用寄存器,R8 ~ R15。

➢ **寄存器用途更改**:

- X64 体系结构下定义了一个可扩展的不可变寄存器(函数调用过程中值被压栈保存起来的寄存器)集合。
- 函数调用时,前 4 个参数通过 RCX、RDX、R8 和 R9 寄存器传递,因此 X64 体系结

构下进行函数调用时，如果参数个数不大于 4 个，则相当于 FastCall 类型。

- RBP 不再用作栈帧寄存器，调试时也不再使用 RBP 回溯堆栈，在 X64 体系结构下 RBP 寄存器是通用寄存器。
- X64 体系结构下使用 GS 寄存器而非 FS 寄存器指向线程的 TEB 或内核的 KPCR，当然在运行 WoW64 进程的时候除外，此时仍按照 X86 体系结构的定义进行指向。
- X64 体系结构下 Trap Frame 自陷框架不再包括不可变寄存器的内容了，需要使用的时候调用序言(Prolog)代码压栈保存，通过堆栈而非 Trap Frame 获取不可变寄存器的内容。

3.1.2　堆栈

"堆栈"由两个词组成，一个是"堆"，一个是"栈"。在系统调用中，我们重点强调的是栈，因此在本文中没有明确说明时，堆栈就专指栈。

栈是一种后进先出的数据结构，用于保存一些使用频繁但是体量小的数据，例如函数的调用框架、自陷框架等。由于后进先出的特性，栈是从高址向低址扩展的(栈的生长方向)，数据进栈的过程叫压栈(push)，出栈就叫弹栈(pop)，栈的底部(高址位置)叫作栈底，低址部分(push 和 pop 发生的位置)叫作栈顶。在 X86 体系结构下，SS 寄存器常用来指示栈底，ESP 寄存器用来指示栈顶，EBP 寄存器用来指示当前栈帧的基址(低址位置)。堆栈结构如图 3-4 所示。

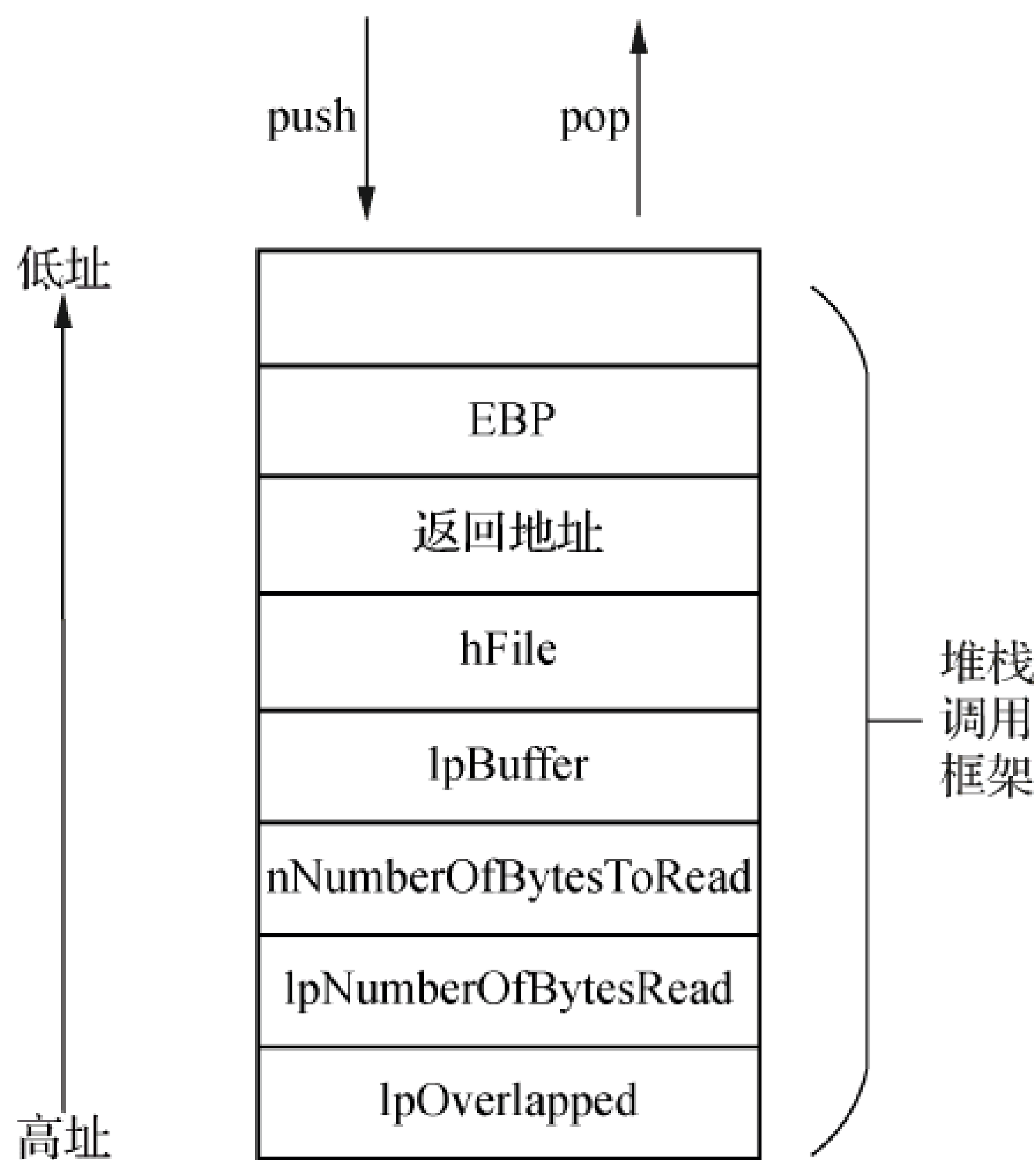


图 3-4　堆栈结构示意图

线程初始化的时候，堆栈大小会有个默认值，例如 4 KB(一个内存页面的大小)。但这并不意味着堆栈大小不能调整，实际上这个 4 KB 的默认大小的内存是提交状态的，还有 4 KB 内存是保留状态的(第二个内存页面)，意味着随时可以投入使用并转入提交状态。当第二个内存页面转入提交状态时，就意味着已经使用到第二个页面了，那就再扩展第三个内存页面为保留状态，如此往复。

这里有必要把线程、函数、堆栈的概念串联起来。函数是一个指令序列，是要被 CPU 执行的实体，而执行的形态就是线程。线程是一个时间序列，函数(指令序列)是在这个(时间序列)中执行的。在执行的过程中可能要传递参数或者保存临时变量(函数调用时构造调用框架)，也可能要保存一些返回值或者寄存器的值(构建栈帧)，此时这些数据总要有个地方存起来，这就是栈。

在栈的生命周期中，一个基本原则叫作配平。也就是说，当调用一个函数的时候，无论是形成栈帧还是调用框架，当这个函数执行完毕时，栈都恢复到调用之前的状态，"好借好还

再借不难”。

用户态线程的生命周期中有可能进行一些系统调用,也有可能自始至终都没有系统调用。前一种情况线程会有用户态和内核态之分,后一种情况自始至终都是用户态。但无论怎样线程都有用户态和内核态两个堆栈,具体位置在 KTHREAD 数据结构中指示。那么为什么要区分用户态堆栈和内核态堆栈呢?在前文介绍过,在 32 位系统中,用户态位于 0x80000000 以下的地址空间,内核态位于 0x80000000 以上的地址空间。在进行系统调用的时候,CPU 状态就从用户态(R3)切换到了内核态(R0)。但是内核态的 CPU 怎么访问用户态的数据(用户态堆栈)呢?难道为了访问一些数据再切回用户态?那还不如干脆再创建一个系统堆栈,岂不是更方便。

3.1.3 GDT 和 LDT

1. GDT

GDT(Global Descriptor Table,全局描述符表)是整个 Windows 系统中全部数据结构的描述符表,与 IDT 一样,每个 CPU 也会有一份对应的 GDT,GDTR 指向 GDT 的物理地址(每个 CPU 有一个 GDTR)。GDT 中的表项叫作全局描述符,每个描述符的长度为 8 字节(64 位),GDT 的总大小为 64K,放置在内存中(地址可以随机,但是需要由 CPU 上的 GDTR 保存),每个 GDT 最多含有 8 192 个表项,其中第一个表项为非法描述符,因此合法的表项最多为 8 191 个。这么多表项 Windows 并没有全都用掉,因此 GDT 后半部分大多是空表项,这些空表项为 GDT 挂钩提供了可能。

下面我们来看一下 GDT 的具体内容,GDT 的结构如图 3-5 所示。

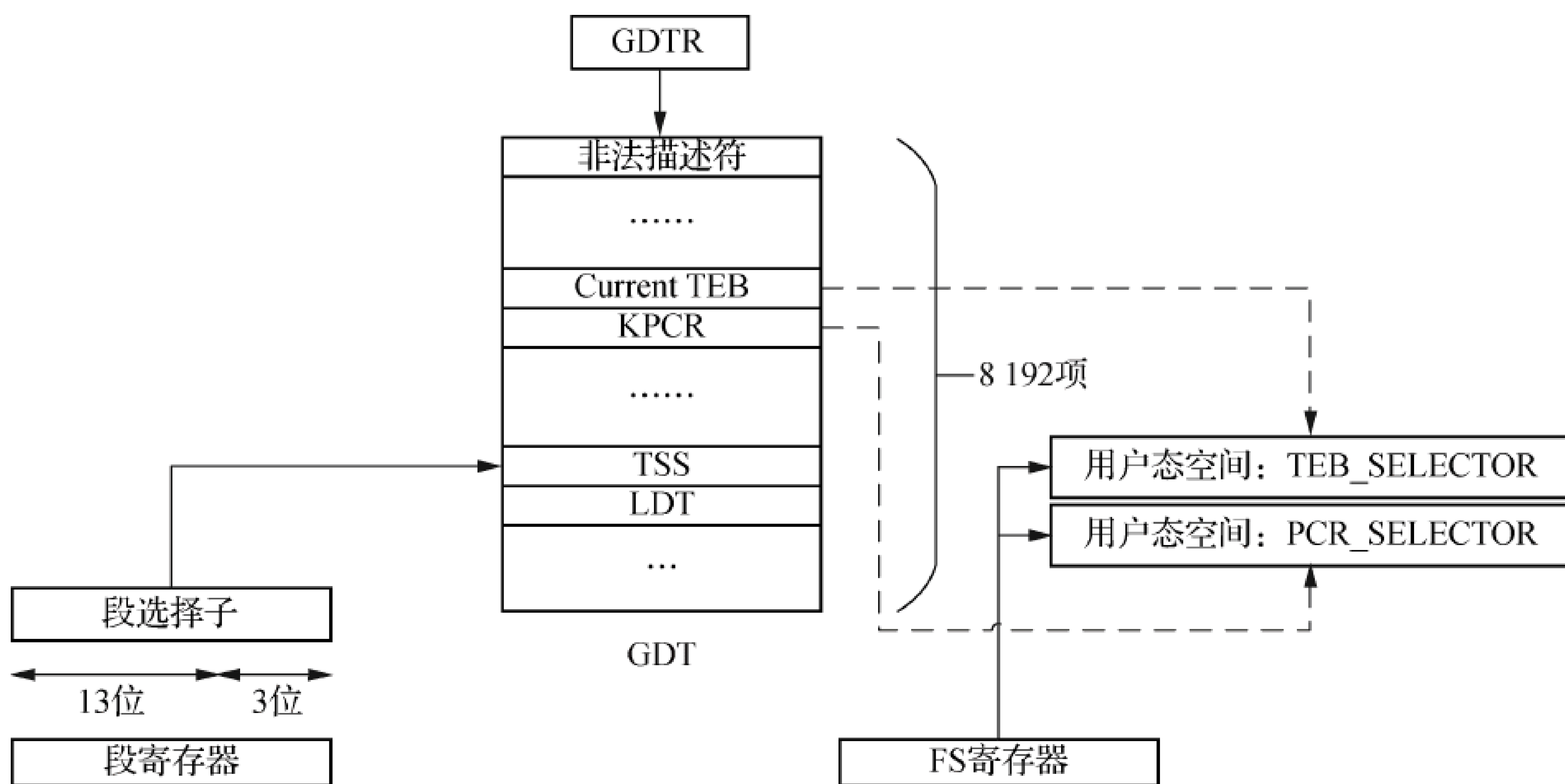


图 3-5 GDT 结构示意图

段选择子(Selector)在段寄存器中为 16 位(可见部分),低 3 位是控制位,表示请求特权级、选择 GDT 还是 LDT 等信息;高 13 位是选择位,表示具体要选择的描述符在 GDT/LDT 中

的索引(从 1 开始),这 13 位正好覆盖了 GDT/LDT 的 8 192 个表项。前文讲过,当 CPU 从 GDT 中读取段描述符的时候,描述符的值会自动写入段寄存器的 64 位不可见部分,以节省再一次访问 GDT 的开销。

FS 寄存器在用户态(R3)的时候指向 GDT 中当前线程的线程环境块(TEB)的选择子:TEB_SELECTOR(KGDT_R3_TEB);当切换到内核态(R0)的时候指向内核处理器控制块 KPCR 的选择子:PCR_SELECTOR(KGDT_R0_PCR)。从 GDT 和 FS 寄存器的指向中可以看出,在 GDT 中当前线程环境块 CurrentTEB 和 LDT 的描述符是个变量,因为随着线程的切换,当前线程的 TEB 和 LDT 必然会变化。由此我们可以推断出,线程切换时 **GDT 中的 TEB_SELECTOR 项会被新的当前线程 TEB 重置**,但事实却不是这样。

原因很简单,线程切换时,TEB_SELECTOR 不变,TEB_SELECTOR 指向的 TEB 只是更换 3 个字段的值,这几个值(SS0、ESP0 和 I/O 权限位图)在每个 TEB 中都不一样。这样做主要是考虑到切换这几个值的总开销远远低于切换 TEB_SELECTOR 的总开销。

我们来直观感受下几个选择子在 32 位 Windows 系统中的定义:

```
#define  KGDT_R0_CODE  0x8     //GDT 中 index =1
#define  KGDT_R0_DATA  0x16    //GDT 中 index =2
#define  KGDT_R3_CODE  0x18    //GDT 中 index =3
#define  KGDT_R3_DATA  0x20    //GDT 中 index =4
#define  KGDT_TSS      0x28    //GDT 中 index =5
#define  KGDT_R0_PCR   0x30    //GDT 中 index =6
#define  KGDT_R3_TEB   0x38    //GDT 中 index =7
#define  KGDT_LDT      0x48    //GDT 中 index =9
```

这里将 LDT 的段选择子限制死了,体现了操作系统设计的灵活性,既可以在 GDT 中保存多个 LDT 的段描述符供进程切换使用,也可以在进程切换时只改变 GDT 中 LDT 的描述符值,不改变 LDTR 中的段选择子。

段描述符分为三种:数据段描述符、代码段描述符、系统段描述符。三种段描述符的差别不是很大,这里就不一一描述了。GDT 中保存着各个全局结构的段描述符,这些段描述符的结构如图 3-6 所示。

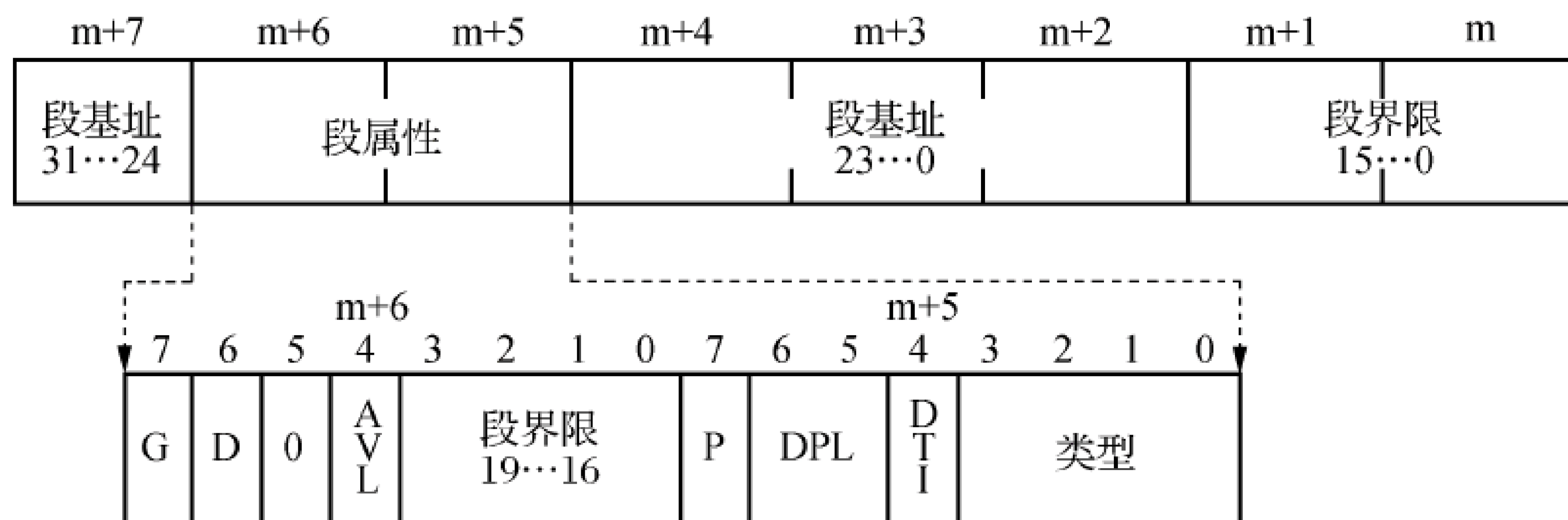


图 3-6　段描述符结构示意图

可以看出,段描述符中除了一些控制位,剩下的就是段基址,段基址 + 段偏移就是我们要访问的线性地址了。

2. LDT

LDT(Local Descriptor Table,局部描述符表)可以有若干个,每个任务可以有一个,也可以没有(任务的概念很抽象,指完成一个活动的载体,达到某一个目的的具体操作,既可以是进程也可以是线程。由于每个进程既有数据段,又有代码段,还有堆栈段,LDT 可以将这些数据集成在一起,只要改变 LDTR 就可以实现对不同进程不同段的访问)。每个 LDT 段描述符的长度也是 8 字节(64 位),LDT 的总大小为 64K,因此每个 LDT 也可以有 8 192 个描述符(有效描述符为 8 191 个)。GDT 是全局性表,因此 GDT 中也含有 LDT 的段描述符,对应了当前进程。同时,LDTR 指向当前任务的 LDT,而不是每个 LDT 有一个 LDTR,而且 LDTR 也是与 CPU 绑定的。因此我们可以这样来看两者的关系:GDT 为一级描述符表,LDT 为二级描述符表。LDT 与任务有关,但不是所有的任务都用到 LDT。当进行任务切换时,处理器会自动把新任务 LDT 的段选择子和段描述符加载进 LDTR 中。

LDT 段描述符的定义与 GDT 一致。访问 LDT 的流程如下:

(1) 从 GDTR 中获取 GDT 基址。

(2) 从 LDTR 中获取要选择的 LDT 的段选择子。

(3) 访问 GDT 中对应 LDT 段选择子的段描述符,该段描述符也同时加载到 LDTR 的 64 位不可见部分。

(4) 从上述段描述符中获取 LDT 段基址,加上段偏移,就是我们要访问的线性地址了。

3.1.4 SSDT 和 IDT

1. SSDT

SSDT(System Services Descriptor Table,系统服务描述符表)是 Windows 系统中最重要的数据结构。Windows 中的系统调用功能都是由 SSDT 来完成的。内核中导出的所有系统调用都在 SSDT 中汇聚和体现,SSDT 基于系统服务编号进行索引以定位函数入口的内存地址。在 Windows 中存在两个 SSDT,一个是由 ntoskrnl. exe 导出的 KeServiceDescriptorTable,另一个是 ntoskrnl. exe 没有导出的 KeServieDescriptorTableShadow(影子 SSDT)。

KeServiceDescriptorTable 仅包括 ntoskrnel. exe 导出的函数,并没有包括 win32k. sys 导出的函数,如图 3-7 所示,也就是说不包含图形相关的系统调用,这部分调用函数(GUI 调用)存放在影子 SSDT 中,常规的非 GUI 的 API 的系统调用(称为原生系统调用)指针由 KeServiceDescriptorTable 分派也就够了,比如由 kernel32. dll 发起的 API 调用等;而 gdi. dll 或 user. dll 等 GUI 动态链接库发起的系统调用指针则由 KeServieDescriptorTableShadow 分派。

实际上 SSDT 并不是仅仅将所有由内核导出的系统调用指针汇聚在一个区域中,而是像一个代理一样把真正的系统调用函数指针的数组地址、参数数组地址等必要的数据形成一个数据结构记录下来。我们以 KeServiceDescriptorTable 为例,SSDT 数据结构如下所示:

SSDT钩子　Shadow SSDT钩子

序号	函数名称	当前函数地址	是否被挂钩	原始函数地址	当前函数所在的模块
0	NtMapUserPhysicalPagesS...	0xfffff80004aa1810			C:\Windows\system32\ntoskrnl.exe
1	NtWaitForSingleObject	0xfffff80004932ae0			C:\Windows\system32\ntoskrnl.exe
2	NtCallbackReturn	0xfffff800046e0d40			C:\Windows\system32\ntoskrnl.exe
3	NtReadFile	0xfffff80004936210			C:\Windows\system32\ntoskrnl.exe
4	NtDeviceIoControlFile	0xfffff80004994dd0			C:\Windows\system32\ntoskrnl.exe
5	NtWriteFile	0xfffff80004936c20			C:\Windows\system32\ntoskrnl.exe
6	NtRemoveIoCompletion	0xfffff80004935c50			C:\Windows\system32\ntoskrnl.exe
7	NtReleaseSemaphore	0xfffff8000494f0d0			C:\Windows\system32\ntoskrnl.exe
8	NtReplyWaitReceivePort	0xfffff80004927070			C:\Windows\system32\ntoskrnl.exe
9	NtReplyPort	0xfffff80004a784c0			C:\Windows\system32\ntoskrnl.exe
10	NtSetInformationThread	0xfffff8000492f860			C:\Windows\system32\ntoskrnl.exe
11	NtSetEvent	0xfffff8000494586c			C:\Windows\system32\ntoskrnl.exe
12	NtClose	0xfffff80004932020			C:\Windows\system32\ntoskrnl.exe
13	NtQueryObject	0xfffff8000494a1d8			C:\Windows\system32\ntoskrnl.exe
14	NtQueryInformationFile	0xfffff80004b01590			C:\Windows\system32\ntoskrnl.exe
15	NtOpenKey	0xfffff800049248a8			C:\Windows\system32\ntoskrnl.exe
16	NtEnumerateValueKey	0xfffff8000492a7a0			C:\Windows\system32\ntoskrnl.exe
17	NtFindAtom	0xfffff800049937b0			C:\Windows\system32\ntoskrnl.exe
18	NtQueryDefaultLocale	0xfffff800049102c4			C:\Windows\system32\ntoskrnl.exe
19	NtQueryKey	0xfffff8000492ec30			C:\Windows\system32\ntoskrnl.exe
20	NtQueryValueKey	0xfffff80004930f10			C:\Windows\system32\ntoskrnl.exe

函数个数：401，被挂函数个数：0

图 3－7　Win7 64 位系统中 ntoskrnl. exe 导出的 KeServiceDescriptorTable

```
typedef struct _SSDT {
   unsigned int * ServiceTableBase;           //ServiceRoutineTable 基址
   unsigned int * ServiceCounterTableBase;    //ServiceRoutineTable 中每个服务被调用次数的
                                                计数器
   unsigned int NumberOfServices;             //ServiceRoutineTable 中有多少个系统调用服务
   unsigned char * ParamTableBase;            //系统服务参数表 SSPT 基址，存放了参数的字节数
}SSDT, * PSSDT
```

其中，ServiceTableBase 才是真正的系统服务表，里面每 4 个字节表示一个系统调用函数的入口基址。在很多杀毒软件的防御体系中要挂钩 SSDT，其实也就是挂钩 ServiceTableBase 中某个具体系统调用的入口。SSDT 的结构如图 3－8 所示。

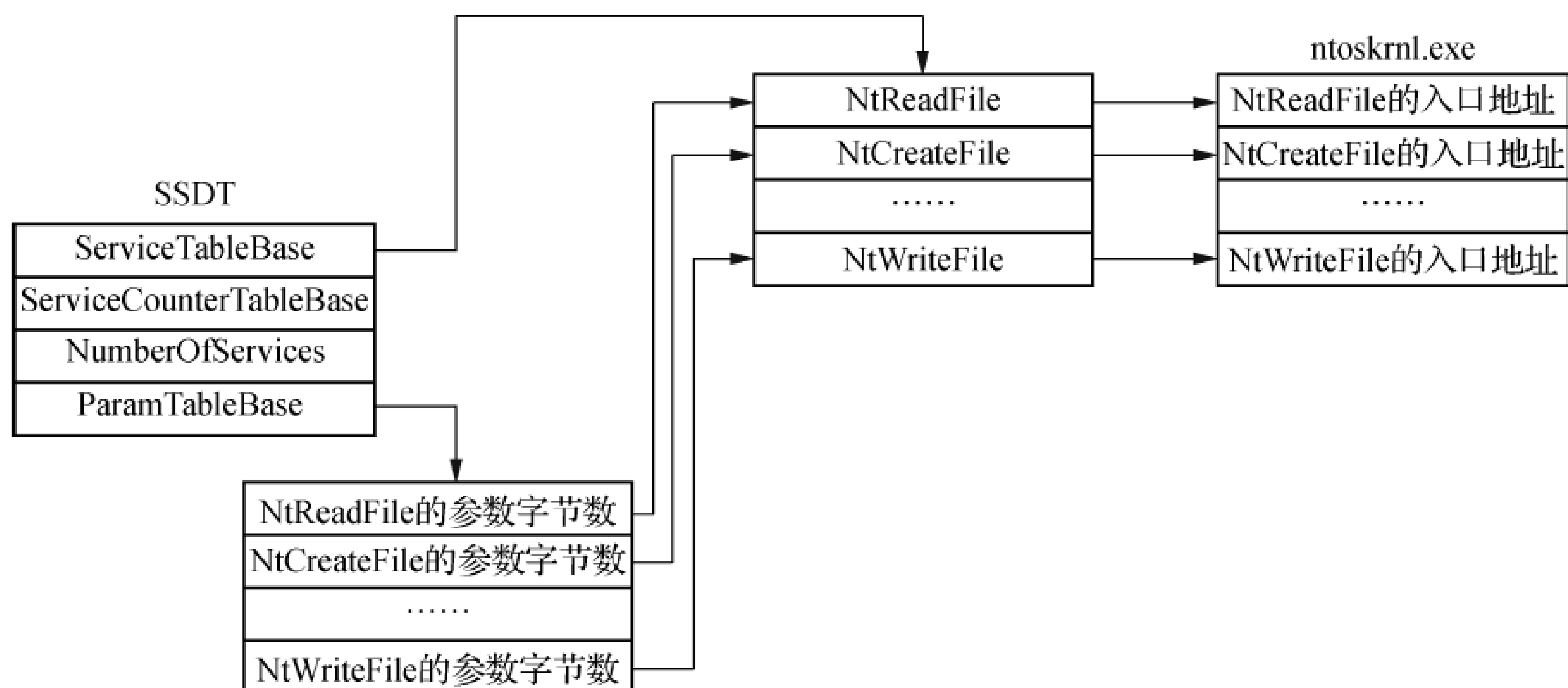


图 3－8　SSDT 结构示意图

KeServieDescriptorTableShadow（影子 SSDT）则包含了 ntoskrnel. exe 和 win32k. sys 两部分的系统调用服务函数入口。具体数据结构是这样的：





```
typedef struct _KeServieDescriptorTableShadow {
  KSERVICE_TABLE_DESCRIPTOR KeServieDescriptorTable;   //KeServieDescriptorTable 的基址
  KSERVICE_TABLE_DESCRIPTOR win32KAPISDT;              //win32k.sys 中 GUI 相关的系统调用
                                                       //的基址
  KSERVICE_TABLE_DESCRIPTOR NotusedTable;              //未用到
  KSERVICE_TABLE_DESCRIPTOR NotusedTable;              //未用到
}KeServieDescriptorTableShadow, * pKeServieDescriptorTableShadow;
```

KeServieDescriptorTableShadow 中的第一项就是 KeServieDescriptorTable 的基址；第二项在 win32k.sys(GUI 相关的系统调用)尚未加载的时候为空，在 win32k.sys 初始化的时候 Windows 调用 KeAddSystemService 将 GUI 相关的系统调用表填充到这个地址；第三、四项未使用。当然，很多的杀毒软件、病毒防御系统、游戏软件也都会挂钩 KeServieDescriptorTableShadow 中的第二项，以方便对系统调用、图形显示等功能进行接管和改造。

Windows 系统对于原生系统调用默认是采用 KeServieDescriptorTable 的，但当当前线程第一次调用 GUI 的接口时，系统会将 KeServieDescriptorTableShadow 作为线程的默认服务表，因此这中间存在一个切换。影子 SSDT 中的函数如图 3－9 所示。

SSDT钩子 | Shadow SSDT钩子

序号	函数名称	当前函数地址	是否被挂钩	原始函数地址	当前函数所在的模块
0	NtUserGetThreadState	0xfffff96000108ec8	-	0xfffff96000108ec8	C:\Windows\system32\win32k.sys
1	NtUserPeekMessage	0xfffff96000105fa8	-	0xfffff96000105fa8	C:\Windows\system32\win32k.sys
2	NtUserCallOneParam	0xfffff96000117970	-	0xfffff96000117970	C:\Windows\system32\win32k.sys
3	NtUserGetKeyState	0xfffff96000125f3c	-	0xfffff96000125f3c	C:\Windows\system32\win32k.sys
4	NtUserInvalidateRect	0xfffff9600011f0d8	-	0xfffff9600011f0d8	C:\Windows\system32\win32k.sys
5	NtUserCallNoParam	0xfffff96000117b74	-	0xfffff96000117b74	C:\Windows\system32\win32k.sys
6	NtUserGetMessage	0xfffff9600010f114	-	0xfffff9600010f114	C:\Windows\system32\win32k.sys
7	NtUserMessageCall	0xfffff960000f331c	-	0xfffff960000f331c	C:\Windows\system32\win32k.sys
8	NtGdiBitBlt	0xfffff960000cb410	-	0xfffff960000cb410	C:\Windows\system32\win32k.sys
9	NtGdiGetCharSet	0xfffff96000220b84	-	0xfffff96000220b84	C:\Windows\system32\win32k.sys
10	NtUserGetDC	0xfffff960001025d4	-	0xfffff960001025d4	C:\Windows\system32\win32k.sys
11	NtGdiSelectBitmap	0xfffff960000fa3c0	-	0xfffff960000fa3c0	C:\Windows\system32\win32k.sys
12	NtUserWaitMessage	0xfffff960001221ac	-	0xfffff960001221ac	C:\Windows\system32\win32k.sys
13	NtUserTranslateMessage	0xfffff96000120b2c	-	0xfffff96000120b2c	C:\Windows\system32\win32k.sys
14	NtUserGetProp	0xfffff9600012458c	-	0xfffff9600012458c	C:\Windows\system32\win32k.sys
15	NtUserPostMessage	0xfffff96000114440	-	0xfffff96000114440	C:\Windows\system32\win32k.sys
16	NtUserQueryWindow	0xfffff960000c5fbc	-	0xfffff960000c5fbc	C:\Windows\system32\win32k.sys
17	NtUserTranslateAccelerator	0xfffff9600011a978	-	0xfffff9600011a978	C:\Windows\system32\win32k.sys
18	NtGdiFlush	0xfffff960000b942c	-	0xfffff960000b942c	C:\Windows\system32\win32k.sys
19	NtUserRedrawWindow	0xfffff96000121538	-	0xfffff96000121538	C:\Windows\system32\win32k.sys
20	NtUserWindowFromPoint	0xfffff96000120e1c	-	0xfffff96000120e1c	C:\Windows\system32\win32k.sys

函数个数：827，被挂函数个数：0

图 3－9　Win7 64 位系统中 KeServieDescriptorTableShadow 中的 GUI 调用

2. IDT

IDT(Interrupt Descriptor Table，中断描述符表)是 Windows 系统中处理中断/自陷的数据结构。诸如除零(0#中断)、断点(3#中断)、早期版本的系统调用(0x2e#中断)以及异常处理等都会引发中断，自然需要相应的中断服务例程进行处理。在 32 位系统中，IDT 共有 256 个表项(门描述符)，每一个表项的长度为 8 字节，因此 IDT 的总大小为 2 KB。IDTR 保存了 IDT 的物理内存地址。每个 CPU 对应一个 IDT，而每一个 CPU 中也有一个 IDTR。

我们先来看一下 IDT 的结构。IDT 中的表项叫作门描述符，一共有三种，其结构如图 3－10 所示。

➢ **任务门描述符**：用于任务切换时保存现场，描述符中包含了 TSS(任务状态段)在

GDT/LDT 中的选择子。(现代操作系统极少使用任务门方式,只是为了兼容 TSS 机制才会在线程切换时更改 TSS 的个别字段,真实的线程切换采用其他方式保存现场。)

- **中断门描述符**:用于描述中断例程入口地址。
- **自陷门描述符**:用于描述自陷处理例程入口地址,例如自陷指令 int 0x2E 对应的服务例程就是 KiSystemService。

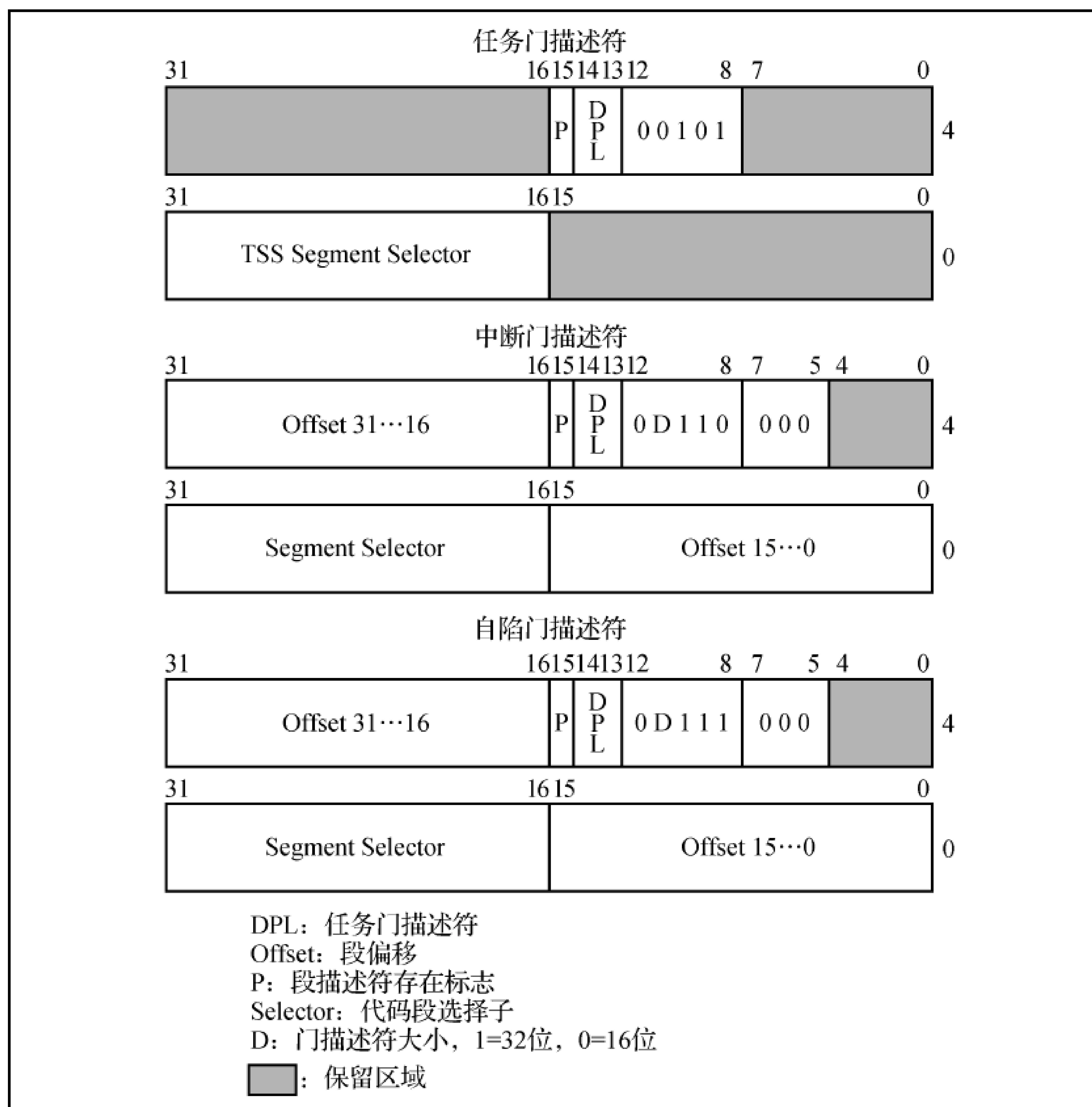


图 3-10　门描述符结构示意图

在早期的 Windows 系统中都是采用自陷(int 0x2E)的方式进行系统调用的,后来的高版本系统(Windows 2000 之后的版本)采用快速调用方式(Intel CPU 采用 sysenter 和 sysexit 指令,AMD CPU 采用 syscall 和 sysreturn 指令)进行系统调用和系统返回,两种调用方式的区别如图 3-11 所示。

由于自陷型系统调用要访问在内存中的 IDT,因此访问速度远比不上快速系统调用方式,稍后的章节中会有详细描述。

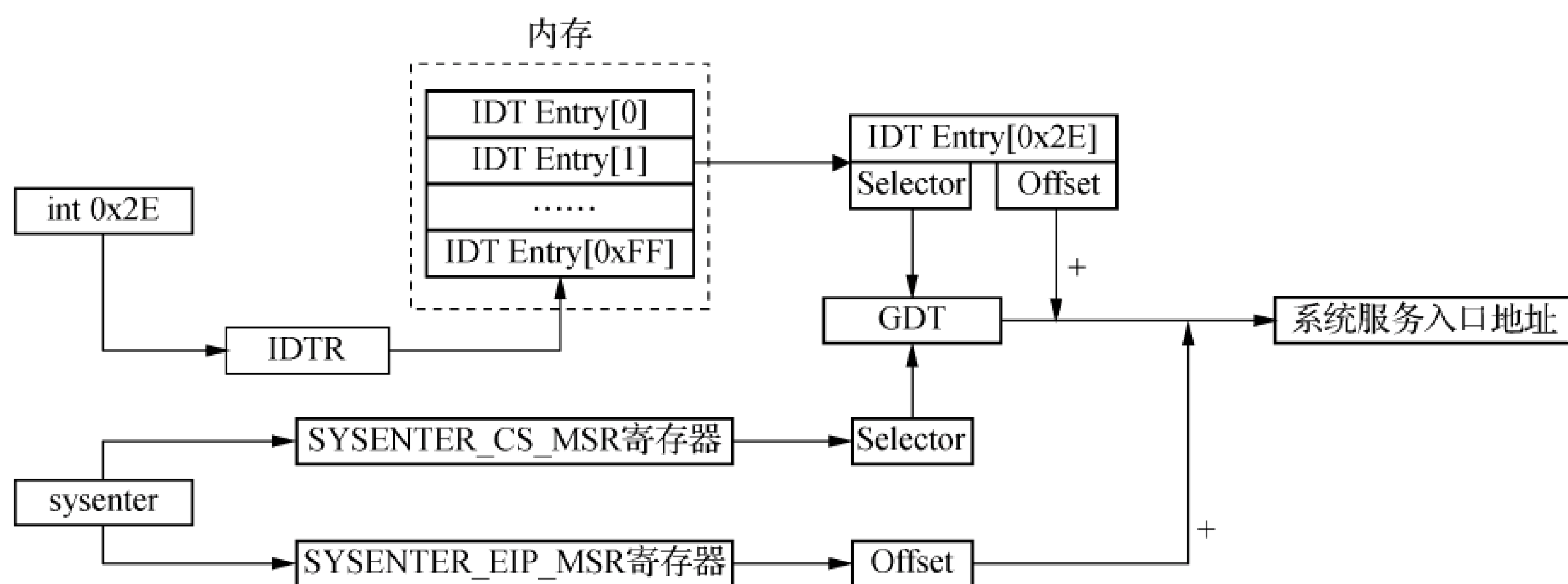


图 3－11　两种系统调用方式的对比

3.1.5　KPCR 和 KPRCB

1．KPCR

所谓 KPCR，就是内核处理器控制区（Kernel Processor Control Region）。前文说过，当 CPU 切换到内核态时，FS 寄存器指向 KPCR 的选择子（在 Windows X64 版本中 GS 寄存器指向 KPCR 的选择子）。从定义就可以看出，KPCR 与每个处理器相对应，描述的是处理器相关的重要数据和状态（例如当前线程信息、GDT 与 IDT 的地址信息等）以及一些线程切换的重要信息。32 位 Windows 系统中 KPCR 在内存中的排布如图 3－12 所示。

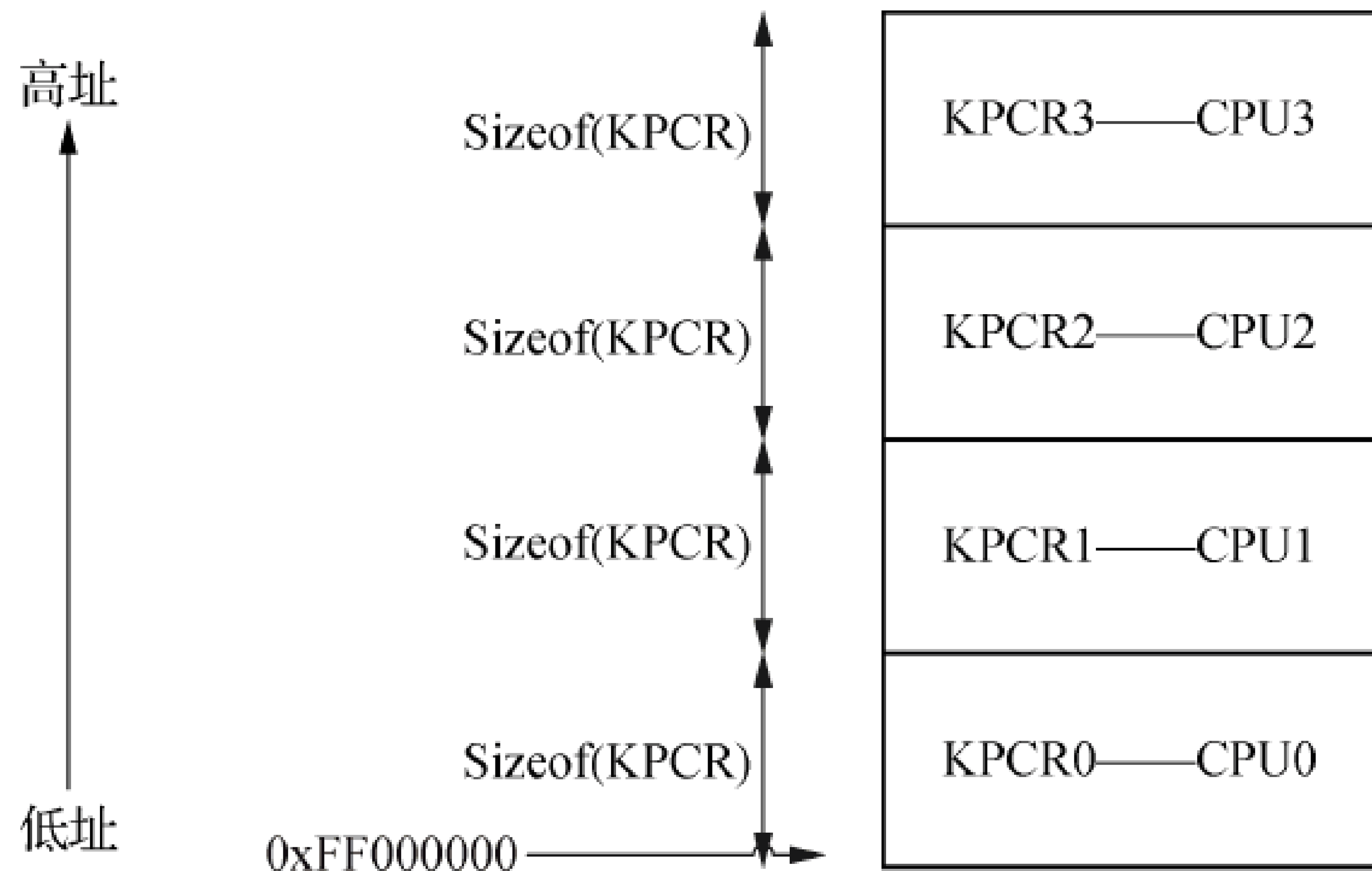


图 3－12　KPCR 在内存中的排布

以下是 KPCR 数据结构的定义：

```
nt! _KPCR
   +0x000 NtTib              :_NT_TIB          //内核态的异常队列结构体
   +0x01c SelfPcr            :Ptr32_KPCR       //指向自己，例如 FS[0x1C]
   +0x020 Prcb               :Ptr32_KPRCB      //指向 KPCR 后面的 KPRCB 数据结构
   +0x024 Irql               :UChar
   +0x028 IRR                :Uint4B
   +0x02c IrrActive          :Uint4B
   +0x030 IDR                :Uint4B
   +0x034 KdVersionBlock     :Ptr32Void
   +0x038 IDT                :Ptr32_KIDTENTRY  //对应 CPU 的 IDT 的线性基址(IDTR 中的值)
   +0x03c GDT                :Ptr32_KGDTENTRY  //对应 CPU 的 GDT 的线性基址(GDTR 中的值)
```

```
+0x040 TSS                :Ptr32_KTSS        //TSS 的线性基址
+0x044 MajorVersion       :Uint2B
+0x046 MinorVersion       :Uint2B
+0x048 SetMember          :Uint4B
+0x04c StallScaleFactor   :Uint4B
+0x050 DebugActive        :UChar
+0x051 Number             :UChar             //对应 CPU 的编号
+0x052 Spare0             :UChar
+0x053 SecondLevelCacheAssociativity:UChar
+0x054 VdmAlert           :Uint4B
+0x058 KernelReserved     :[14]Uint4B
+0x090 SecondLevelCacheSize:Uint4B
+0x094 HalReserved        :[16]Uint4B
+0x0d4 InterruptMode      :Uint4B
+0x0d8 Spare1             :UChar
+0x0dc KernelReserved2    :[17]Uint4B
+0x120 PrcbData           :_KPRCB            //对应的内核处理器控制块
```

2. KPRCB

KPRCB(Kernel Processor Control Block,内核处理器控制块)本质上是对 KPCR 相当大幅度的扩展,其中很多字段用于线程切换和调度,其指针位于 KPCR 的最后一个。KPRCB 与 KPCR 一一对应,自然也与 CPU 一一对应。由于其数据结构很大,我们在这里只是选择性地列出几项,如下所示:

```
nt! _KPRCB
......
 +0x004 CurrentThread    :Ptr32_KTHREAD    //当前线程结构体 KTHREAD 的指针
 +0x008 NextThread       :Ptr32_KTHREAD    //下一个需要切换到的线程结构体 KTHREAD 的指针
 +0x00c IdleThread       :Ptr32_KTHREAD    //当 CPU 没有其他线程执行时要切换到的空闲线程结
                                           //构体 KTHREAD 的指针
......
 +0x4a8 KernelTime       :Uint4B           //内核态执行时间,用于统计信息
 +0x4ac UserTime         :Uint4B           //用户态执行时间,用于统计信息
......
WaitListHead             :_LIST_ENTRY      //线程切换时,等待线程的队列
ReadySummary             :  ULONG          //就绪队列的位图
DispatcherReadyListHead  :[32]_LIST_ENTRY  //32 个优先级的继续队列
```

3.1.6　KTHREAD、ETHREAD、W32THREAD 和 TEB

前文讲过,Windows 内核最上面两层分别是执行体层和内核核心层。执行体层负责管理、策略等事务,而更底层的实现机制是由内核核心层负责的。线程作为操作系统执行的基本单位,在执行体层和内核核心层分别有数据结构予以对应,分别是 ETHREAD 和 KTHREAD 结构,在用户态则使用 TEB(线程环境块)结构表示。W32THEWAD 结构处于内核核心层,是为视窗进程所属的线程准备的,包含了与图形界面窗口相关的数据和属性。

由于内核核心层涉及线程调度和切换,因此 KTHREAD 结构的很多域都与切换和调度有关。TEB 结构包含了在用户态空间中需要访问的各种数据,而 ETHREAD 结构中则包含了 APC 机制、进程挂靠等管理和策略信息。

这里,我们不准备长篇大段地讲述这几个数据结构,而且这几个结构确实庞大,不太适合集中讲述。我们在此只是简要地提一下,在后面的篇幅中若有用到它们的地方会详细阐述。对这些数据结构有兴趣的读者可自行查阅相关资料。

3.1.7　TSS

TSS(Task State Segment,任务状态段)在 GDT 中也有相应的段描述符,TR 指向当前使用的 TSS 物理地址。在最初设计时,Windows 操作系统希望在进行线程切换时 TSS 用于保存当前现场(例如系统中各寄存器的值,以方便进行线程切换,每次切换时只更换 TSS,其他的寄存器值就不单独切换了)。但实际上线程切换时只需要更换 TSS 中的个别值,且更换个别值的执行开销远小于替换整个 TSS,并且由于各线程的 KTHREAD 等数据结构相当完善,因此将线程切换的现场保存工作寄希望于整个 TSS 就显得没有必要。在实际切换线程时,无论是在 Windows 系统中还是在 Linux 系统中,都只需要改变 TSS 的 SS0、ESP0 和 I/O Map Base Address 字段(分别是当前线程的内核态堆栈选择子、堆栈指针和 I/O 权限位图)。

TSS 数据结构如图 3－13 所示。

31	15　　0	偏移
I/O 位图基址	T	100
	LDT 段选择子	96
	GS	92
	FS	88
	DS	84
	SS	80
	CS	76
	ES	72
EDI		68
ESI		64
EBP		60
ESP		56
EBX		52
EDX		48
ECX		44
EAX		40
EFLAGS		36
EIP		32
CR3(PDBR0,页目录表基址寄存器)		28
	SS2	24
ESP2		20
	SS1	16
ESP1		12
	SS0	8
ESP0		4
	上一任务链接	0

图 3－13　TSS 数据结构示意图

线程切换时,TSS 的变化是:从 TR 获取 TSS,更新 I/O 权限位图、ESP0、SS0,由于所有线程的内核态堆栈选择子都是一样的,因此具有实际意义的变化是前两项,如图 3－14 所示。除此之外,就是 GDT 中的 TEB 和 LDT 要更新一下了。

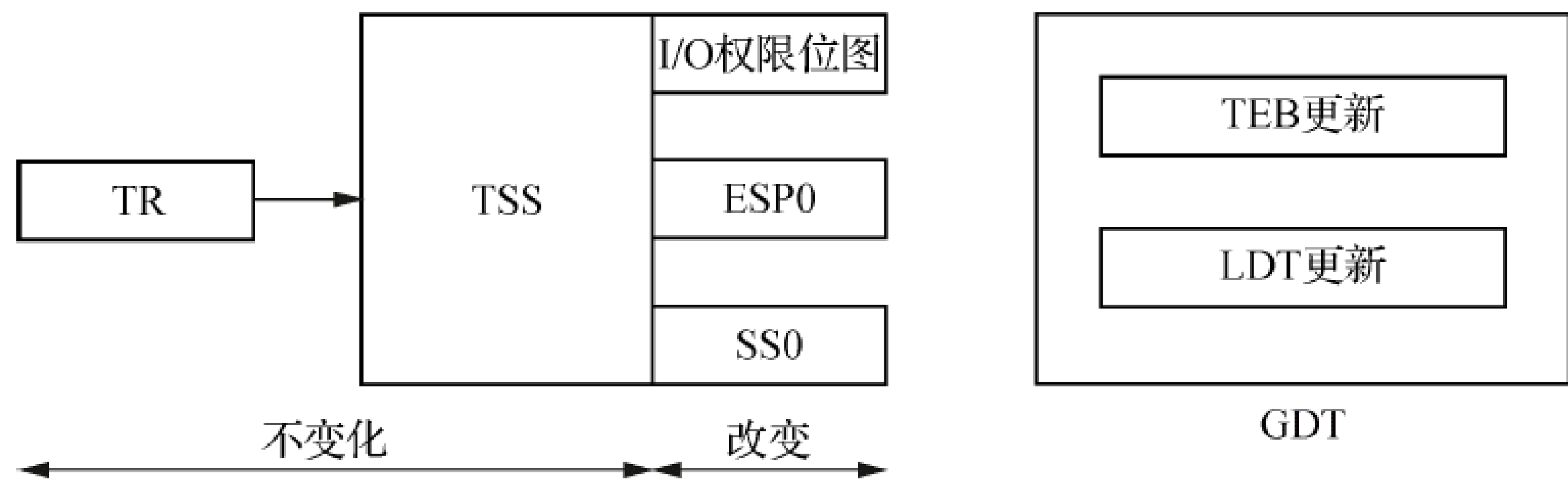


图 3－14　线程切换时 TSS 的变化

以上各小节中对于预备知识中细节的讲解并不是完全到位的，在后面的章节中，我们还会根据实际需要进行针对性的细节描述。

3.1.8　调用约定

函数的调用约定是指函数被调用时，参数的传递方式和栈的配平策略（由谁来进行栈的配平和清理）。调用约定可以根据 CPU 是 X86 架构的还是 X64 架构的分为两类，如表 3－1 所示。

表 3－1　调用约定描述

CPU 架构	调用约定种类	参数入栈顺序	栈清理者（pop 以配平栈）	其他说明
X86	_cdecl	从右至左入栈	调用者（caller）清理栈区	C/C++ 的默认调用约定
	_stdcall	从右至左入栈	被调用者（callee）清理栈区	Windows API 的默认调用约定
	_fastcall	从右至左入栈，前两个参数分别被放入 ECX、EDX 寄存器	被调用者（callee）清理栈区	如果不多于两个参数，会先将参数放入寄存器
	_pascal	从左向右入栈	被调用者（callee）清理栈区	
	_thiscall	从右至左入栈，this 指针存放于 ECX 寄存器	对参数个数不定的，调用者（caller）清理堆栈；否则，由被调用者（callee）清理堆栈	仅用于 C++ 类成员函数，如果参数不确定，this 指针在所有参数被压栈后压入栈堆
X64	_fastcall	从右至左入栈，前 4 个参数分别被放入 ECX、EDX、R8、R9 寄存器，更多的参数放入栈区	调用者（caller）清理栈区	系统为前 4 个参数预留了栈区空间，每个栈空间的大小为 8 字节，寄存器的值被放入这 4 个位置，以备寄存器接收其他值而无法传参

从上表可以看出，虽然 X64 架构下函数调用的前 4 个参数被放在了寄存器中，但是栈帧空间仍然为这 4 个值分配了空间（8B ×4），这种特性叫作“Homing Space”，但只有 X64 架构下的 non-leaf（非叶子节点）函数才会有这个特性。所谓 non-leaf 函数就是函数执行过程中还要调用其他函数的函数，而 leaf 函数自然就是执行过程中不再调用其他函数的函数。

在 Homing Space 中，如果不存放参数的值，编译器会将其改用作存放不可变寄存器的值。所谓不可变寄存器，就是函数调用过程中值被保存起来的寄存器。X64 平台拥有一个扩展的不可变寄存器集合，包括 X86 架构下固有的不可变寄存器加上 R12 ~ R15 四个寄存器。

在 X64 架构下的函数调用过程中，调用框架也发生了细微的变化。X64 堆栈中的调用框架由返回地址（Return Address）、不可变寄存器的值（Non Volatile Registers）、局部变量（Local Variables）、基于栈的参数（Stack Parameters）和基于寄存器的参数（Register Parameters）构成，其中基于寄存器的参数至少要保留 32 个字节，如图 3－15 所示。

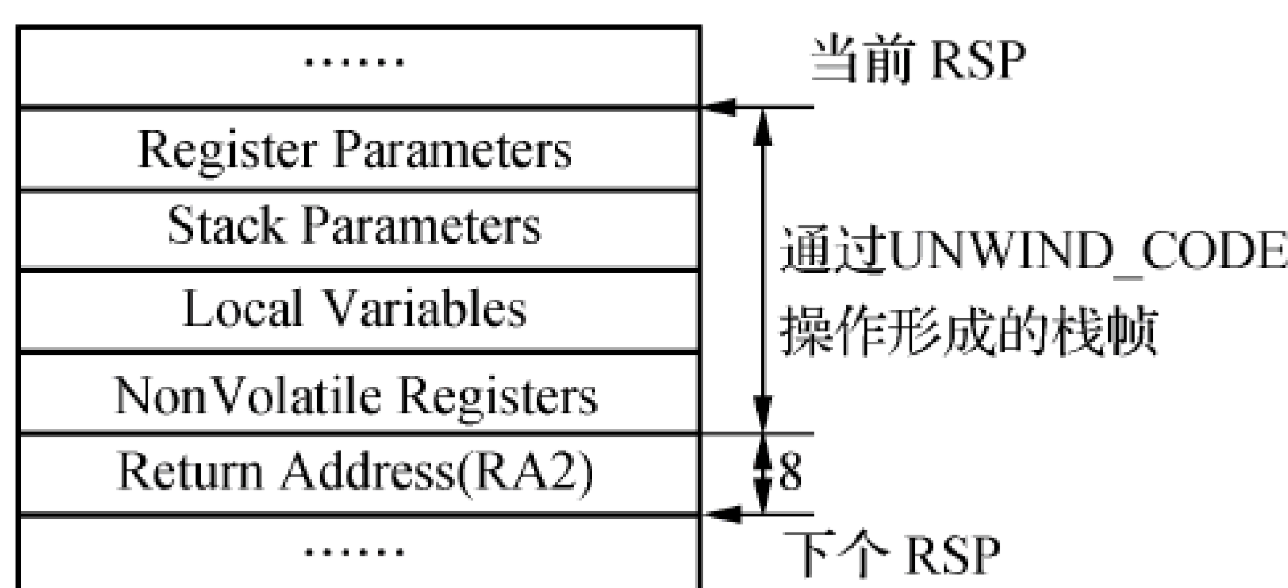


图 3－15　X64 架构下的函数调用框架

X64 架构下，函数调用过程还做了以下修改：

1）消除尾部调用（Tail Call Elimination）

X64 编译器可以使用 jump 指令替换函数体内最后部分的 call 指令，这样做有以下优势：

- 避免被调函数创建栈帧。
- 调用函数和被调函数可以共享相同的栈帧。
- 被调函数可以直接返回到自己父函数的父函数，这在调用函数和被调函数参数相同的情况下具有很高的执行效率。

2）栈帧指针省略（Frame Pointer Omission，FPO）

X86 和 X64 架构下的 FPO 是有细微差异的，在 X64 架构下使用 RSP 作为栈帧基址寄存器和栈顶寄存器，RBP 只作为通用寄存器使用，不但节省了一个寄存器，也加快了栈帧的访问速度。

3）基于栈顶指针的局部变量访问（Stack Pointer Based Local Variable Access）

由于 X64 架构下 RSP 寄存器作为栈帧基址寄存器，因此依赖于 RSP 的函数栈帧在函数体执行过程中是不能改变 RSP 寄存器的值的，故而也会限制 push 和 pop 指令的执行，即只能在函数的首尾更改、使用，中间的执行过程只能读而不能写，如图 3－16 所示。

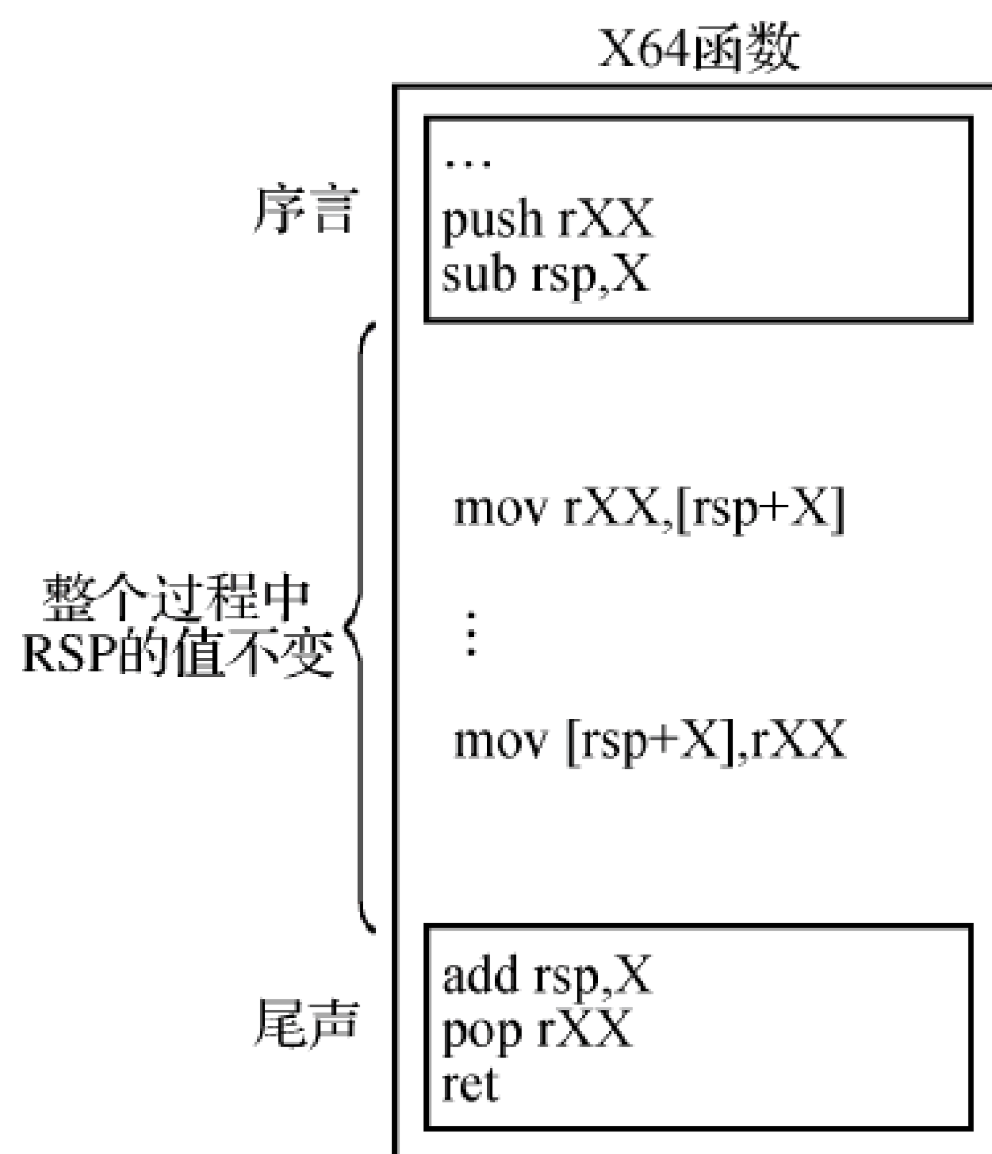


图 3－16　X64 函数执行过程中对于 RSP 寄存器的使用

3.2　自陷型系统调用流程

所谓自陷型系统调用，是指系统调用是通过 IDT 中的自陷中断服务例程（int 0x2E 指令）实现的。这中间要访问主存中的 IDT、GDT 等多个数据结构，因此运行速度要比快速型系统调用慢（快速型采用新增的专有寄存器实现）。下面我们先介绍自陷型系统调用的总体切换流程，继而拆解整个流程的执行序言、执行跳板、执行尾声等过程细节，是一个从切换到返回的过程描述。

3.2.1　切换流程

我们以 Win32 API ReadFile 为例，结合流程图和堆栈示意图，来讲解系统调用的具体流程。本质上系统调用最核心的部分就是堆栈切换以及堆栈内容的此消彼长（配平），只要明白了堆栈的数据消长也就明白了系统调用的核心流程。

（1）用户态软件调用 kernel32. dll 中的 Win32 API ReadFile，ReadFile 又调用 ntdll. dll 中的 ZwReadFile，故事就从这里开始了。此时，该 API 还运行在用户态空间，其堆栈状况如图 3 – 17 所示。

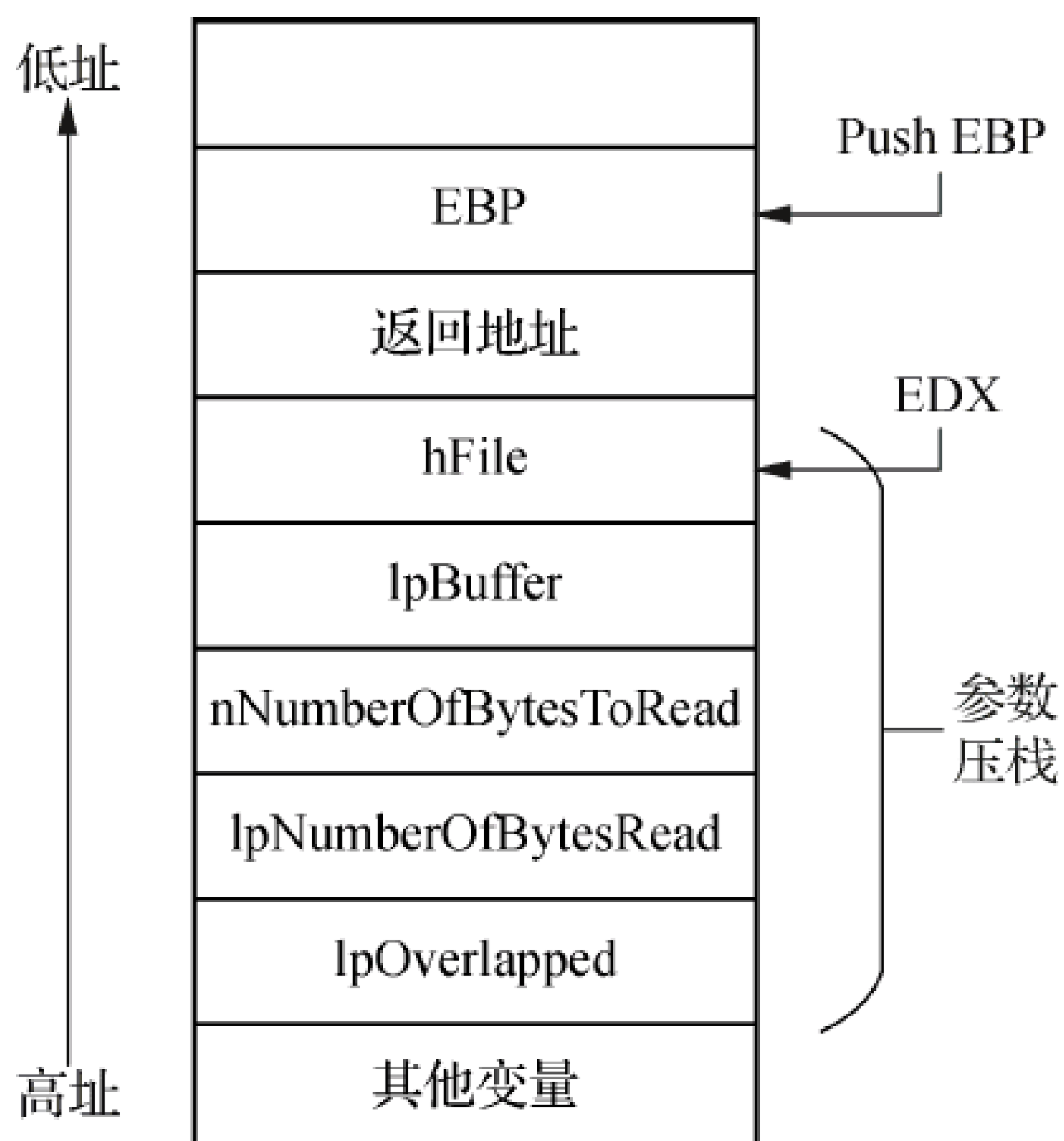


图 3 – 17　用户态堆栈结构示意图

可以看到用户态栈从高址到低址依次压入了 ReadFile 的 5 个参数。注意，参数是倒序压栈的，即函数的最后一个参数先压栈，第一个参数最后压栈。之后会把 ReadFile 的返回地址压栈。所谓返回地址，是指紧接着 ReadFile 的整个汇编指令块后面的一条汇编指令，即汇编指令 ret 的下一条指令的地址（ret 是各个函数汇编指令块的最后一条指令，代表了该函数已经全部执行完毕，可以返回了。返回到哪里？自然是返回到 call 函数这条汇编指令的下一条）。最后将 EBP 指针压入堆栈，此时函数的调用堆栈栈帧构筑完毕。EBP 指向某个栈帧的低址起始位置（X86），即某栈帧的底部（相对低址高低来说的底部，而非栈的底部，这里要分清楚）。采用 EBP 可以比较方便直观地访问当前栈帧中的变量。

这里简单说一下 Nt 和 Zw 接口的区别。ntdll. dll 导出的接口中既有 Nt 前缀也有 Zw 前缀（都是存根函数），以 ReadFile 为例，在用户态调用 ZwReadFile 和 NtReadFile 没有区别，只是相同函数的不同名称而已。在内核态调用时就不能用 ntdll. dll 了，这时需要使用内核态的 ntoskrnl. exe 导出的接口，例如 ZwReadFile 和 NtReadFile。Zw 前缀函数也叫存根函数（stub），它执行的时候会将线程的 Previous Mode（先前模式）设置为 Kernel Mode，然后调用 NtReadFile，而 NtReadFile 是直接执行系统调用的，就是 SSDT 中的系统调用，并不做先前模

式的更改。

(2) 在调用 ntdll. dll 的 ZwReadFile 的过程中,事情慢慢发生了变化。ZwReadFile 首先会将 Readfile 的系统调用号(152)赋值给 EAX 寄存器,之后会执行指向用户态空间的一个地址块 KUSER_SHARED_SYSCALL 的代码,这个地址块在 CPU 进行初始化的时候也同步进行初始化,要么被赋值为自陷型中断函数入口,要么被赋值为快速调用型系统接口(后面介绍),详见本节末尾的注释。我们以前者为例,自陷型中断函数入口是 KiIntSystemCall,而 KiIntSystemCall 又会触发 int 0x2E 指令,int 0x2E 执行自陷并切换堆栈。从这一步可以看出,ntdll. dll 才是从用户态向内核态切换的“转换开关”。

(3) int 0x2E 首先从 TR 中获取 TSS 结构,从 TSS 的 SS0 和 ESP0 字段中获取当前线程的内核态堆栈底部和顶部参数,并赋值给 SS 寄存器和 ESP 寄存器。注意,此时堆栈才真正从用户态切换到内核态。想想也是,一个数据段的选择子和地址都换了,那不就是“改朝换代”了嘛。

(4) 切换了堆栈,要向新的堆栈(系统堆栈)依次压入 5 个参数,即用户态 SS、ESP、EFLAGS、CS、EIP。其实这就是用户态的上下文,也就是说,有了这些值,可以随时恢复当前线程的用户态堆栈、状态标志和代码被打断执行时的位置。

(5) int 0x2E 还要进行一个大动作,即通过 IDTR 获取 IDT 的基地址,继而获取该系统调用中断号(0x2E)对应的中断描述符表项 IDTEntry[0x2E],并从中获取选择子(Selector)和偏移量(Offset)。Selector 用来从 GDT/LDT 中选择系统代码段(例如 KGDT_R0_CODE)选择子并赋值给 CS 寄存器,Offset 直接定位到当前的执行指令(例如 KiSystemService)的位置并赋值给 EIP 寄存器。执行完毕,线程在内核态中的堆栈地址数据和指令数据都已备齐。

(6) CPU 开始执行 EIP 寄存器中的指令,即执行 KiSystemService,这是自陷型系统调用的总入口。而该函数要执行以下 4 个步骤:执行序言、执行跳板、执行系统调用、执行尾声。

图 3 - 18 以堆栈段和代码段的构筑过程详细描述了系统调用的前期流程。注意,这里只是前期流程,描述了这么多,其实当前的工作只是完成了:

➢ 用户态堆栈向内核态堆栈的切换(数据段切换)。

➢ 向内核态堆栈写入用户态线程当前的执行上下文(保存现场)。

➢ 用户态执行代码段向内核态执行代码段的切换(代码段切换)。

下面就要具体执行 KiSystemService 了,也就是前述步骤的最后一步。这一步要执行前奏、执行跳板、执行系统调用、执行尾声,最后系统调用返回。我们将在下一小节详细描述。

注:X86 系列 CPU 从 Pentium II 开始增设了三个专有寄存器(Model Specific Register)和两条专用指令,用来支持快速调用。它们分别是 SYSENTER_CS_MSR 寄存器(编号 0x174)、SYSENTER_EIP_MSR(编号 0x176)寄存器和 SYSENTER_ESP_MSR(编号 0x175)寄存器,以及 sysenter 和 sysexit 指令。而对于这三个寄存器的读写也只能由专门的特权指令 rdmsr 和 wrmsr 来完成。三个寄存器分别用来存储内核态的堆栈段指针、代码段指针和当前执行指令指针,具体如下:

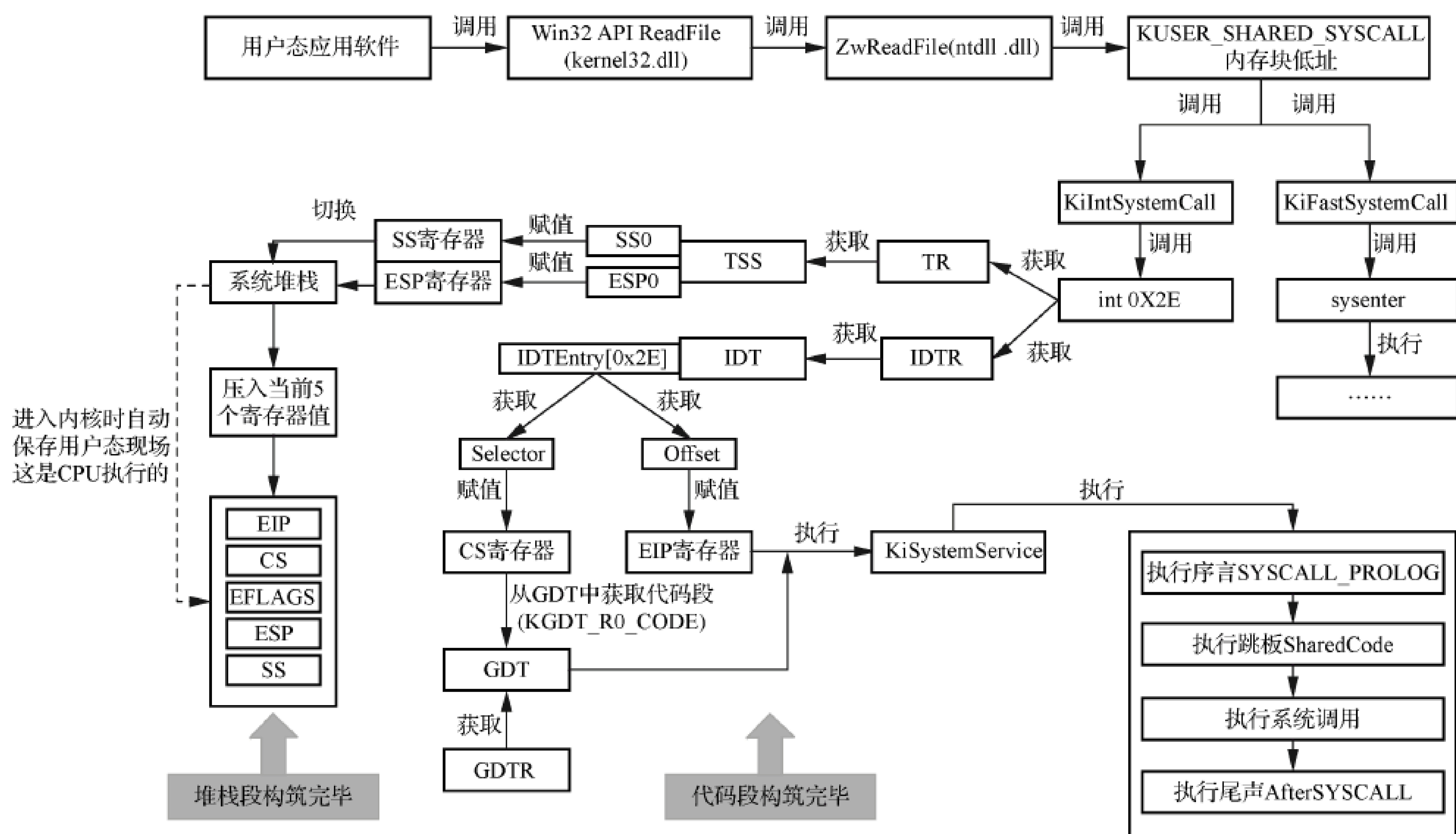


图 3－18　自陷型系统调用流程图

- SYSENTER_CS_MSR 寄存器存放**内核态代码段选择子**。
- SYSENTER_CS_MSR 寄存器中的内容(内核态代码段选择子)加 8(在 GDT 中的描述符索引加 1),这个地址存放**内核态堆栈段选择子**。
- SYSENTER_CS_MSR 寄存器中的内容(内核态代码段选择子)加 16,这个地址存放**用户态代码段选择子**。
- SYSENTER_CS_MSR 寄存器中的内容(内核态代码段选择子)加 24,这个地址存放**用户态堆栈段选择子**。
- SYSENTER_EIP_MSR 寄存器存放**内核态代码指针**。
- SYSENTER_ESP_MSR 寄存器存放**内核态堆栈指针**。

调用号是一个 32 位的序号,如图 3－19 所示代表了系统调用函数在 SSDT 中的索引。但这 32 位不都是有效的,只有低 12 位(第 0～11 位)才表示了对应调用在 SSDT 中的索引,而第 12 位则代表服务表号,0 表示选择 SSDT,1 表示选择 Shadow SSDT。无论是自陷型系统调用还是后面要描述的快速型系统调用,调用号都是使用 EAX 寄存器来承载的。

19位：未使用	1位：服务表号	12位：系统服务索引号

图 3－19　调用号格式

快速调用方式下的 sysenter 和 sysexit 指令并不依赖内存,只依赖专有寄存器。而内核态的堆栈段选择子、代码段选择子以及堆栈指针和指令指针都可以从专有寄存器中获取,不需要访问内存,从而提高了执行速度。快速调用方式下 GDT 中的每个表项依然为 8 字节,内核态代码段描述符、内核态堆栈段描述符、用户态代码段描述符、用户态堆栈段描述符按顺

序依次排列。

在系统进行初始化的时候,系统会根据 CPU 的型号决定 KUSER_SHARED_SYSCALL 这块内存到底是装入自陷型系统调用入口(KiIntSystemCall)还是快速型系统调用入口(KiFastSystemCall)。

3.2.2 执行序言

所谓序言,就是执行系统调用之前的准备工作,包括在内核态堆栈中构建自陷框架、寄存器内容切换和重定向、内核线程块 KTHREAD 的内核态初始化等。首先我们来看内核态堆栈的情况。前面我们曾经描述过系统堆栈已经有了 5 个值,即用户态线程的运行现场,我们称之为“保存现场”。以此为基础,接下来我们要构造自陷框架。所谓自陷框架,就是保存了状态切换时各个寄存器以及线程状态属性的堆栈段。我们来看图 3-20。

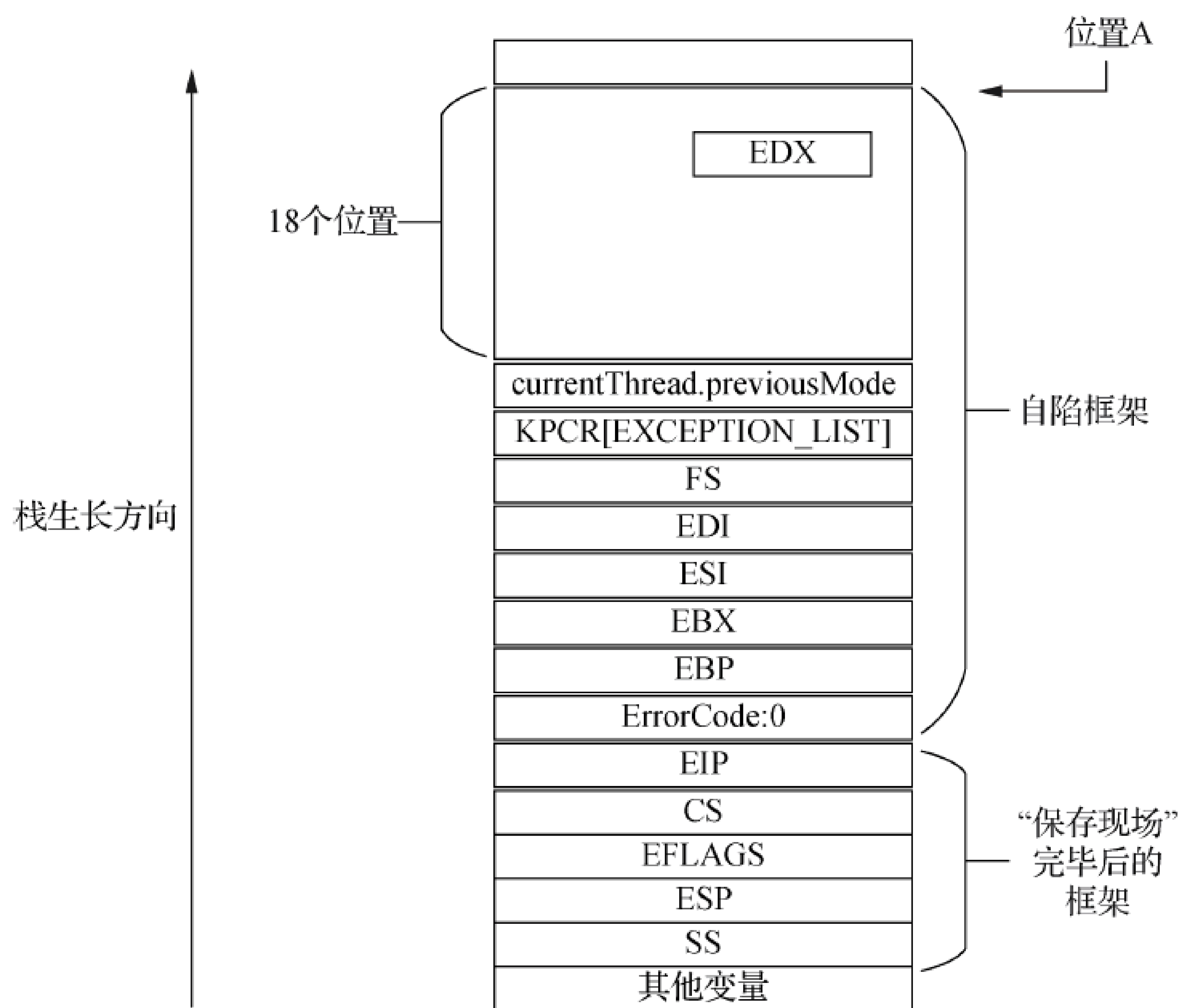


图 3-20 执行序言后的系统堆栈示意图

可以看到,堆栈底部是堆栈段初始化完毕后的框架,即用户态线程现场上下文。之后开始构造自陷框架。自陷框架先压入一个 ErrorCode 作为占位,一般是 0,之后将各个寄存器的值按顺序压入栈内,其中 KPCR[EXCEPTION_LIST]的位置指向老的异常队列。关于 PCR 前几节已经有描述。这里,位置 A 处以上(栈生长方向的反方向)有 18 个位置,即 0x48 个字节,用于保存 EAX、EDX 等寄存器的值。这里面需要强调的是 EDX 寄存器,它用于保存前一次进入中断或者自陷时的自陷框架 KTRAP_FRAME 指针(低址指针),即每一次进入自陷或中断,都会将上一次的 KTRAP_FRAME 指针保存在这个位置,从而形成嵌套堆栈链。我们平时用 Windbg 等工具进行栈回溯的时候就是基于这种原理。

在执行序言阶段，一些寄存器也要重定向或者重新赋值：

- FS 寄存器进行重定向切换，指向 KGDT_R0_PCR（PCR_SELECTOR）。
- ESI 寄存器指向当前线程的 KTHREAD。
- KPCR[EXCEPTION_LIST]被赋值为 0xFFFFFFFF。
- 当前线程的 KTHREAD 的 Previous Mode 被赋值为用户模式（如果是从用户态进入内核态）。
- 当前线程的 KTHREAD 的 Trap Frame 被赋值为位置 A 的地址，即赋值为新的自陷框架的指针。
- EBP 寄存器指向位置 A，表示这是新的自陷框架的起点。

至此，自陷型系统调用的序言序列已经执行完毕。总结一下，其主要工作就是构造自陷框架，对一些寄存器进行重定向切换，对当前线程的某些属性进行重置和更新。

3.2.3　执行跳板

跳板，也叫 Shell Code。执行跳板的主要目的是将用户态堆栈中的函数参数复制到内核态堆栈中，如图 3－21 所示，并且使 EDX 寄存器指向 SSDT 中具体的服务例程入口。

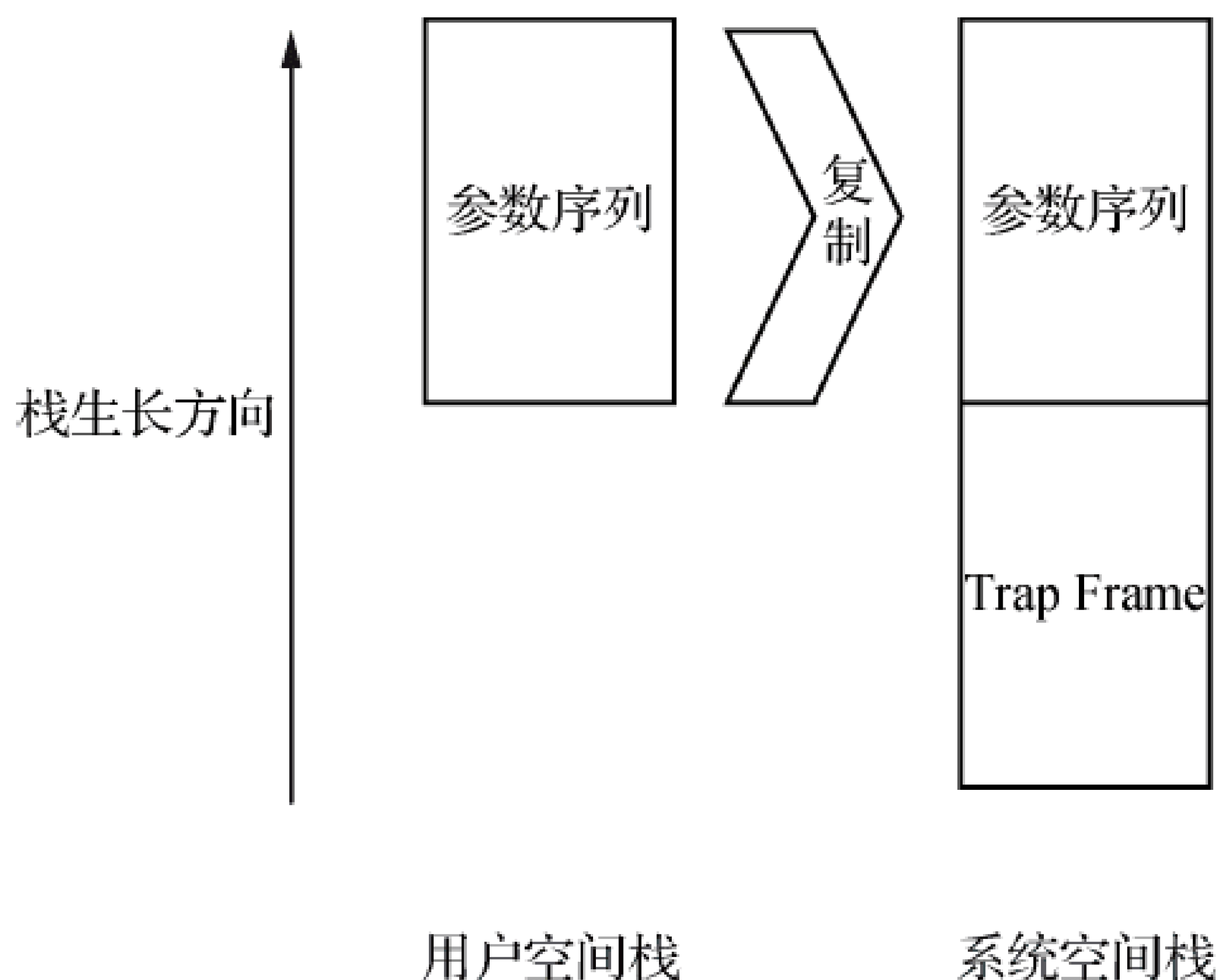


图 3－21　从用户态堆栈复制参数

具体到 ReadFile，则 EDX 寄存器指向 KeServiceDescriptorTable[0].base[152]。这里的 152 是 NtReadFile 在 SSDT 中的函数表下标。那么 KeServiceDescriptorTable 又是怎么来的呢？原来 KTHREAD 数据结构的 ServiceTable 域中存放着本线程的系统调用服务表指针，这个指针或者指向 KeServiceDescriptorTable，或者指向 KeServiceDescriptorTableShadow。若是前者，则只支持基本的原生系统调用；若是后者，则既支持基本系统调用，也支持 GUI 相关调用。

执行完跳板后，EDX 寄存器中已经是即将执行的函数的入口了，该函数的参数可以从内核态堆栈获得，调用时执行一个 call 指令就可以了。

3.2.4　执行尾声

执行尾声是为了处理函数返回后的堆栈配平、寄存器还原、返回用户态空间等一系列操作，基本上是执行序言的逆过程。

（1）当前 EBP 寄存器中的值为图 3－20 中的位置 A 的地址，即本次调用形成的自陷框架基址。因为既然执行到尾声，则系统调用肯定已经执行完了，自陷框架之上（栈生长方向）的调用框架就失去了作用，所以直接将 EBP 寄存器的值赋给 ESP，表示回到序言时的堆栈状态。

（2）自陷框架中 EDX 的位置，即 ESP 指针＋KTRAP_FRAME_EDX 这个位置保存着上次自陷的框架指针，将其还原回当前线程的 KTHREAD 的自陷框架地址上。

（3）执行 KiServiceExit，即检查 APC 队列并执行，然后执行 TRAP_EPILOG，这就是 TRAP_PROLOG/SYSCALL_PROLOG 的逆过程，以消除内核态堆栈中的自陷框架。APC 队列的处理我们在后面章节中会详细介绍，这里不展开。

（4）消除自陷框架后，在尾声执行过程中将内核态堆栈中的剩余变量弹出。若调用来自内核态空间，则将 EIP 的值弹出到 EDX 寄存器中，并 jmp 到 EDX 寄存器，由于都是在内核态空间，因此堆栈段的地址指针不需要切换。若调用来自用户态空间，则在弹出这 5 个变量后执行 iret 指令，将 EIP、CS、SS、ESP、EFLAGS 这 5 个寄存器的值出栈，并弹出到对应的寄存器中，这一步是恢复用户态现场。由于 SS 和 ESP 寄存器的变化，当前执行环境切换回用户态堆栈并返回用户态空间代码段，接下来执行 EIP 就可以了。

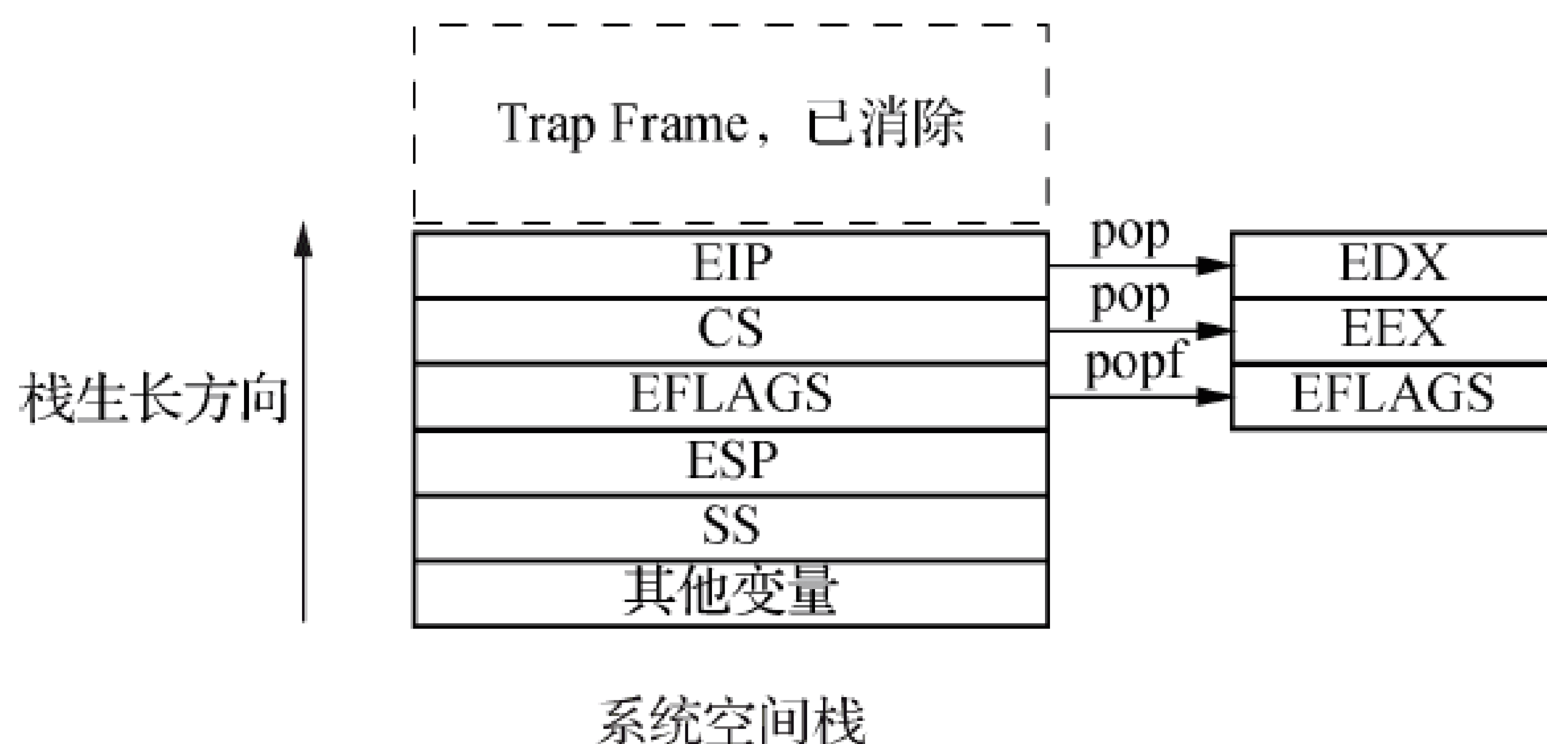


图 3－22　执行尾声时内核态堆栈示意图

至此，系统调用尾声执行完毕，内核态堆栈配平。

注：上文中提到了 APC 机制，在这里我们简要介绍下 APC 的执行时机，后面 APC 相关章节会有详细介绍。

➢ 内核 APC 的执行时机：
- 每次返回用户态空间前（KiServiceExit）；
- 执行 KeLowerIrql（从 APC_LEVEL 以上降级到 APC_LEVEL 或以下）时；
- 线程重新调度开始运行前，会扫描执行内核 APC 队列或发出 APC 中断请求。

➢ 用户 APC 的执行时机：从内核态返回用户态之前（例如系统调用、中断、异常等）。

3.3　快速型系统调用流程

快速型系统调用是指利用专有寄存器保存系统调用的相关数据结构,避开对内存中 IDT 的访问而直接获取调用入口的方式。我们以自陷型系统调用为参照讲解这个过程。

3.3.1　切换流程

我们仍以 ReadFile 为例来介绍快速型系统调用的流程。如图 3 – 23 所示是快速型系统调用的主要流程,可以看出它比自陷型系统调用要简单很多,特别是从 sysenter 指令开始,到用户态向内核态切换完成,快速型系统调用流程大大简化,由于选择子都存放在专有寄存器中,因此不需要访问 TSS 或 IDT 便可填充 CS 等寄存器,访问速度大大加快。

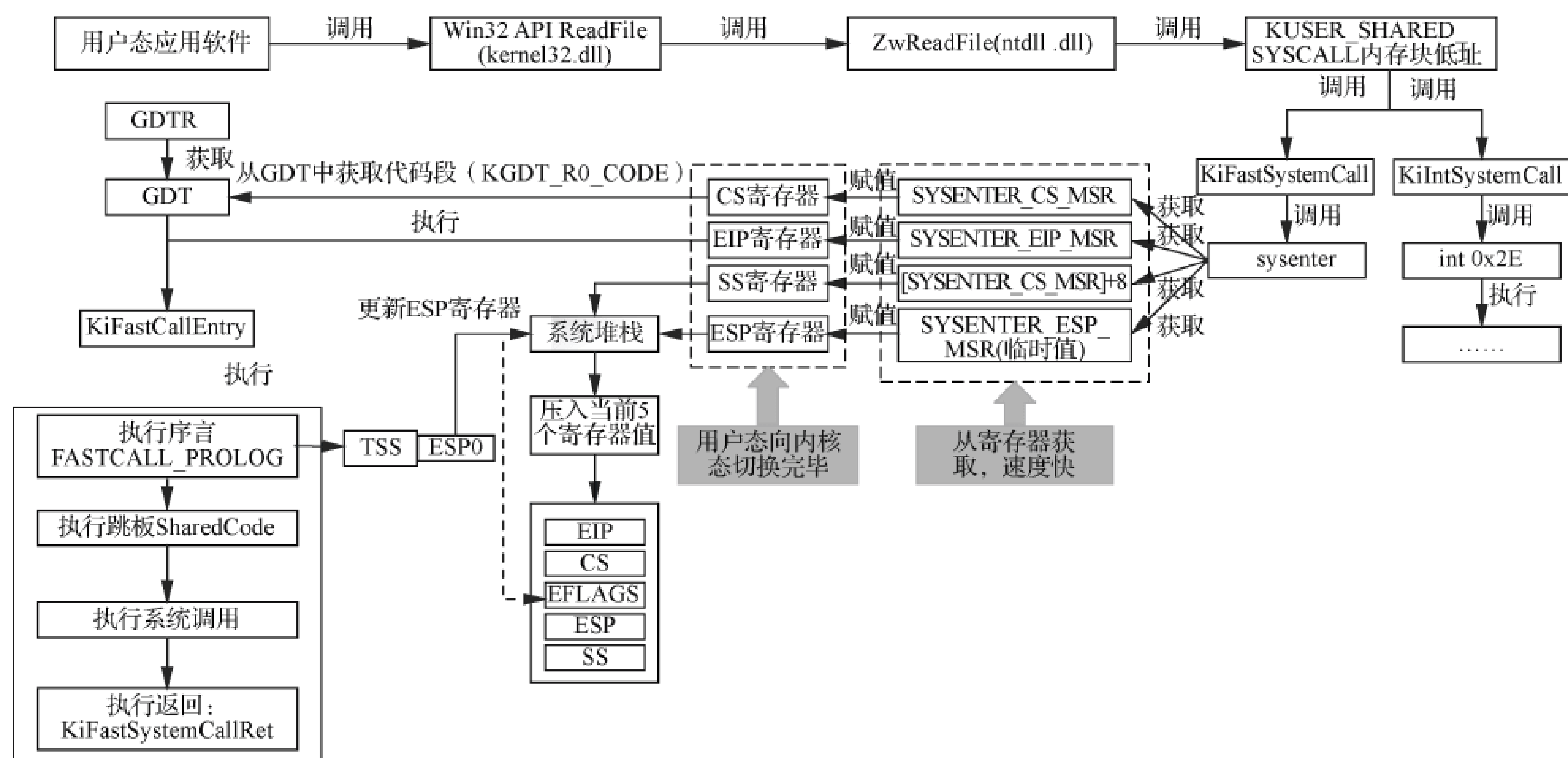


图 3 – 23　快速型系统调用流程图

在快速型系统调用流程中,切换这一步相当于自陷型系统调用流程中的切换、执行序言和执行跳板这三步,其具体步骤如下(仍以 ReadFile 为例):

(1) 用户态线程调用 kernel32 中的 Win32 API ReadFile,ReadFile 又调用 ntdll. dll 中的 ZwReadFile。通过用户态的 KUSER_SHARED_SYSCALL 内存块(保存了 KiFastSystemCall 入口)执行 KiFastSystemCall,继而调用 sysenter 指令(Intel CPU 环境下)。但线程此时尚在用户态,使用的也是用户态堆栈。这里要将 ESP 寄存器的值赋予 EDX 寄存器,如此 EDX 寄存器的值加 8(沿用户态堆栈回溯 2 个位置)的位置就是用户态堆栈参数块的位置,便于将参数块拷贝到内核态堆栈。

(2) 执行 sysenter 指令,具体如下:

➢ sysenter 指令从 SYSENTER_CS_MSR 寄存器中获取内核态代码段选择子(KGDT_R0_CODE)并赋值给 CS 寄存器;

- 从 SYSENTER_EIP_MSR 寄存器中获取内核态代码段函数入口指针(KiFastCallEntry)并赋值给 EIP 寄存器;
- SYSENTER_CS_MSR 中的内核态代码段选择子加 8 就是内核态堆栈段选择子(KGDT_R0_DATA),将其赋值给 SS 寄存器,这也客观上要求它们在 GDT 中按顺序相邻排布;
- 从 SYSENTER_ESP_MSR 寄存器中获取内核态堆栈指针并赋值给 ESP 寄存器(是个临时值,在序言中会进行实际赋值)。

这些寄存器被赋值后,用户态向内核态的切换完成。

(3) 执行 KiFastCallEntry。首先执行快速调用的序言 FASTCALL_PROLOG,在内核态堆栈中构造自陷框架。与 int 指令不同的是这里的用户态上下文的保存是序言代码中实现的,而 int 指令则先保存上下文再执行前奏。各寄存器和当前线程的 KTHREAD 的更新如下:

- FS 寄存器进行重定向切换,指向 KGDT_R0_PCR。
- ESP 寄存器更新为 TSS 中的 ESP0。
- ESI 寄存器指向当前线程的 KTHREAD。
- KPCR[EXCEPTION_LIST]被赋值为 0xFFFFFFFF。
- 当前线程的 KTHREAD 的 Previous Mode 被赋值为用户模式(如果是从用户态进入内核态)。
- 当前线程的 KTHREAD 的 Trap Frame 被赋值为位置 A 的地址(如图 3-24 所示),即填充新的自陷框架的位置。
- EBP 寄存器指向位置 A,表示这是自陷框架的起点。

可以看出,快速型与自陷型系统调用的序言在保存现场之后的步骤完全一致。自陷框架的构建参见图 3-24 中的内核态堆栈,其中最左边的堆栈是自陷型系统调用的系统堆栈,中间的是快速型系统调用的系统堆栈,可以看看两者的区别:

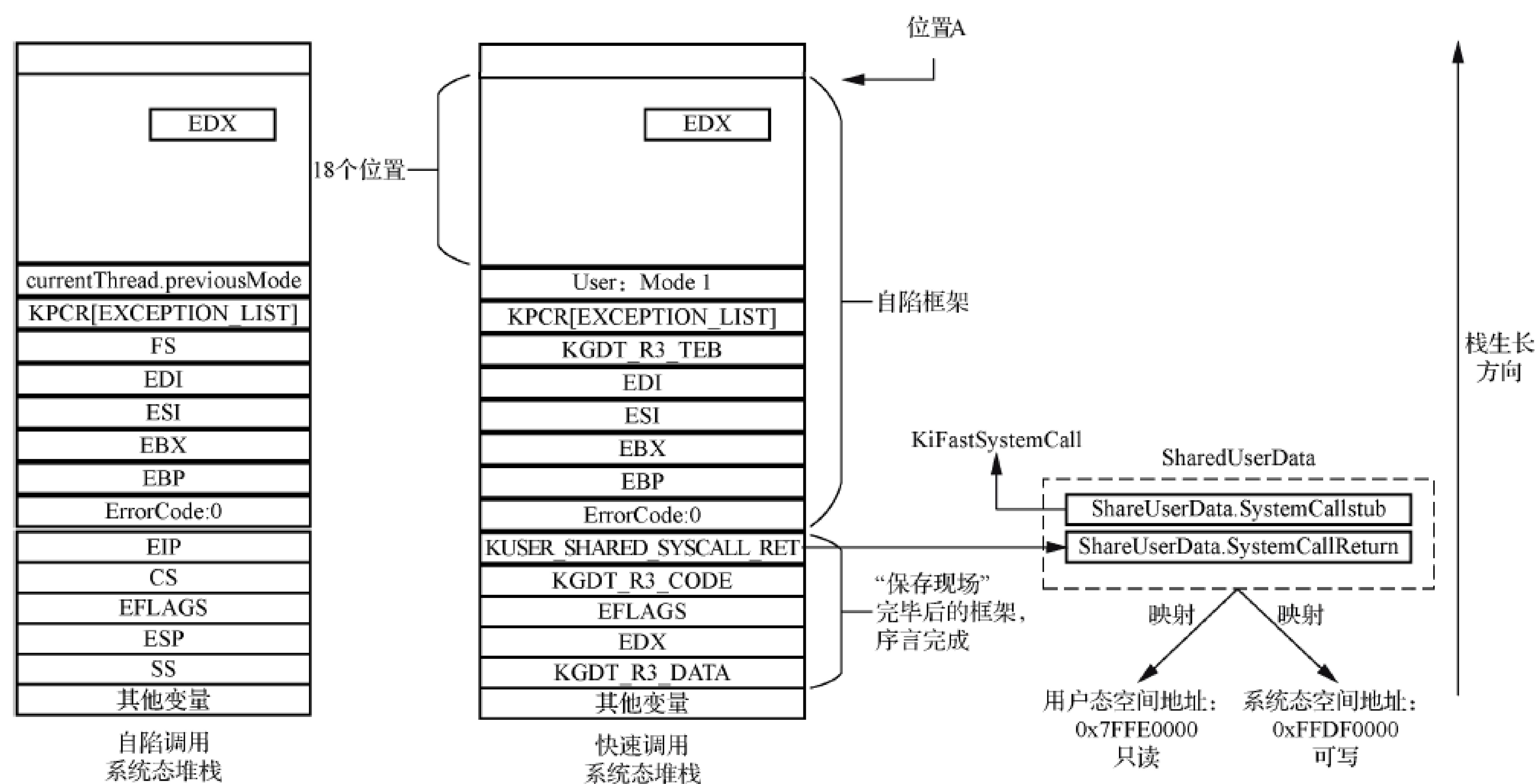


图 3-24　两种系统调用的状态切换堆栈示意图

- 原来 SS 寄存器的位置现被赋值为 KGDT_R3_DATA，表示这是通用用户态堆栈段选择子。
- 原来 ESP 寄存器的位置现被赋值为 EDX 寄存器的值，因为在步骤（1）的时候已经将用户态的 ESP 寄存器的值赋给 EDX 寄存器了。
- 原来 CS 寄存器的位置现在被替换为 KGDT_R3_CODE，表示这是通用用户态代码段选择子。
- 原来 EIP 寄存器的位置现在被替换为 KUSER_SHARED_SYSCALL_RET，这个值指向用户态空间中一块共享数据区域的某个位置，实际上就是 KiFastSystemCallRet，表示系统调用的返回总入口。
- 原来 FS 寄存器的位置现在指向 KGDT_R3_TEB，即指向 GDT 中当前线程的 TEB 选择子。
- 先前模式的位置改为 1，即表示先前模式为用户态模式。

综上所述，上述两种调用方式中堆栈里的内容略有更改，但是其含义和栈帧大小完全相同。这里我们要说明一下用户态的 SharedUserData，它在用户态空间和内核态空间被分别映射，也就是“一块内存两次映射”，在用户态只读，在内核态可写，因此初始化的时候操作系统会在内核态进行写入。这块内存的 SystemCallStub 入口就是 KiFastSystemCall 或 KiIntSystemCall。

执行跳板（Shared Code），这里的 Shared Code 与自陷型系统调用中的基本一致，不再赘述。

3.3.2　执行返回

返回函数的主要任务有：

- 消除自陷框架。
- 将 FS 寄存器重新指向 KGDT_R3_TEB。
- 将 KUSER_SHARED_SYSCALL_RET 弹出到 EDX 寄存器中。
- 将堆栈中保存的 EDX 的值（这是之前保存的用户态堆栈指针）弹出到 ECX 寄存器中。
- 执行 sysexit 指令，具体如下：
  - 将[SYSENTER_CS_MSR] + 16 赋值给 CS 寄存器，[ ]表示取值，也就是将 KGDT_R0_CODE + 16（KGDT_R3_CODE）赋值给 CS 寄存器。
  - 将[SYSENTER_CS_MSR] + 24 赋值给 SS 寄存器，也就是将 KGDT_R0_CODE + 24（KGDT_R3_DATA）赋值给 SS 寄存器。
  - 将 EDX 寄存器的值（KUSER_SHARED_SYSCALL_RET）赋值给 EIP 寄存器。
  - 将 ECX 寄存器的值（用户态堆栈指针）赋值给 ESP 寄存器。

执行完成后，内核态堆栈重新切换为用户态堆栈，最后执行 EIP 就可以了。

本章小结

系统调用是包括 Windows 在内的任何操作系统最重要的部分,进程要实现的大多数功能都是通过系统调用实现的。同时,系统调用一般也是用户态和内核态的分水岭。本章首先对 Windows 系统调用相关的各种寄存器和数据结构进行介绍,继而梳理了两种系统调用类型:自陷型系统调用和快速型系统调用。

在自陷型系统调用中以堆栈变化和平衡为主线讲述了状态切换的步骤、前奏的执行、跳板的执行和尾声的执行。在快速型系统调用中依然以堆栈动态变化和平衡为主线描述了状态切换和执行返回的步骤。

第 4 章　进程与线程的创建

CPU 执行的基本单位是线程，而进程是线程的创建者和容器。作为创建者，进程为线程提供了必要的数据结构（包括 KTHREAD、ETHREAD、TEB、用户态堆栈、内核态堆栈等）和执行的“初始动能”——启动线程执行；作为容器，进程为线程提供了一切必备的执行环境，包括进程虚拟内存空间、虚拟内存空间布局等，这些也是创建进程过程中的必备动作。

进程创建线程的过程具有一定的技巧性：

- 首先进程会在线程内核态堆栈中构造一些伪数据结构，造成线程从这里被剥夺执行权的假象，方便线程开始执行的时候有初始参数。
- 然后借用 IRQL（中断请求级别）降低的档口执行异步过程调用（APC）回到用户态加载其余依赖的模块。
- 通过 APC 返回用户态空间时根据线程的参数启动线程的初始执行入口函数。

本章将按照图 4－1 所示的提纲进行介绍。

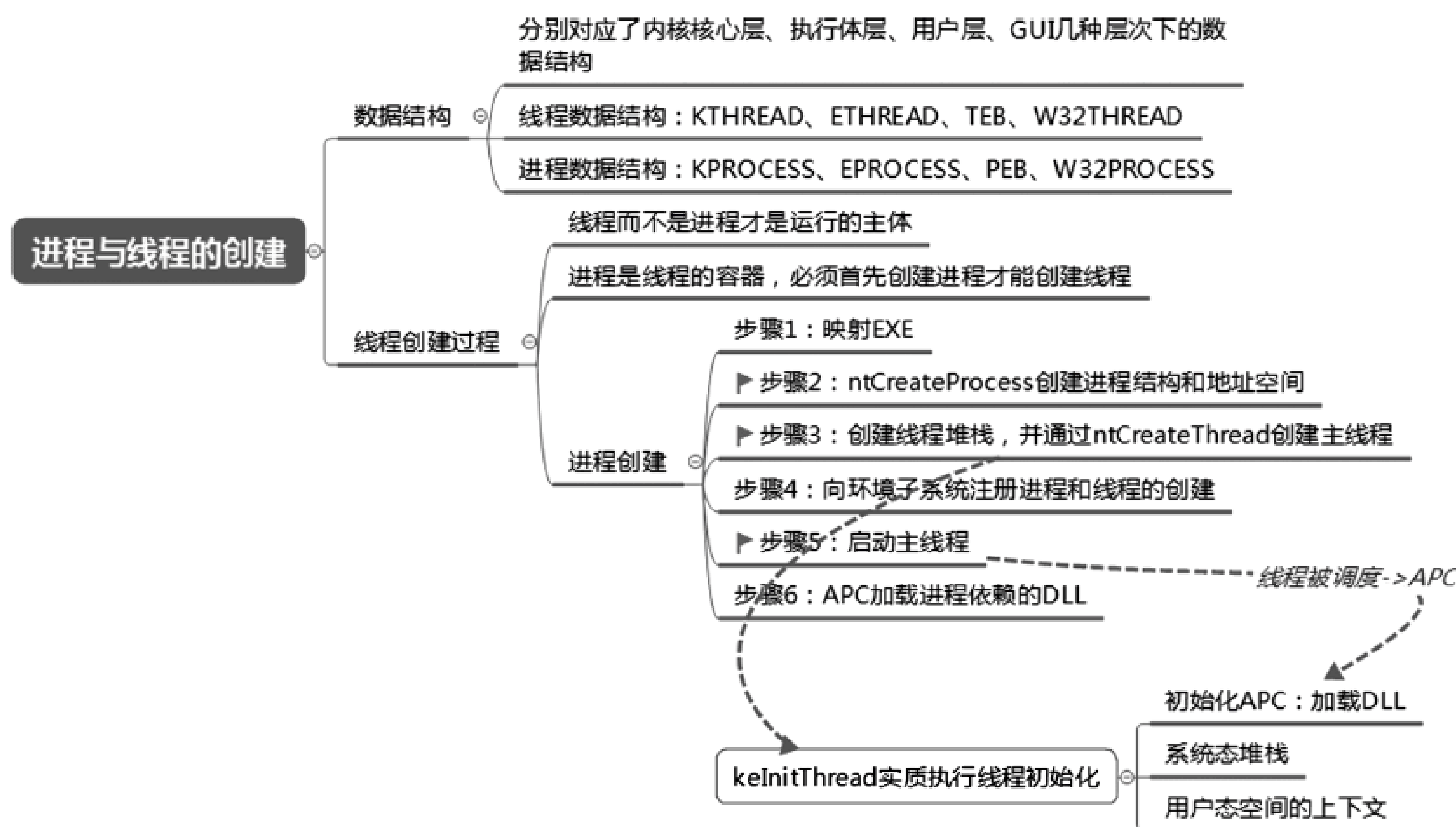


图 4－1　本章提纲

4.1　数据结构

我们有必要先来看一看进程和线程的相关数据结构。在前文中曾简单描述过线程相关的数据结构（KTHREAD、ETHREAD、W32THREAD、TEB 等），这里我们会较为详细地描述它们之间的关系。前文铺垫过，线程和进程在操作系统的不同层次都有对应的数据结构来体现，因为这些不同的层次具有不同的核心任务，对于进程/线程数据结构的要求也不一而足，

将这些要求隔离在不同的数据结构中看起来是明智之举。当然,每个数据结构都不会只呈现自己在操作系统对应层次中需要承担的那些任务,也会附带一些其他的数据,这是因为操作系统是个很复杂的系统机制,每个层次会用到的数据并不固定,数据结构之间会有交叉关联。限于篇幅,我们不会枚举这些数据结构的每个字段,完整的数据结构读者可以自行查阅相关资料。

4.1.1 线程相关数据结构

"KTHREAD"中的"K"即 Kernel(核心),KTHREAD 就是线程在内核核心层的数据结构,如图 4-2 所示,也叫作线程控制块(TCB),TCB 自然位于内核态,主要负责线程调度相关事宜。

Dispatch Header//等待时使用的结构体，该结构体是个分发器对象，可以挂入等待队列
……
Kernel Stack//指向系统态空间堆栈
……
APCState//表示APC的状态
……
TEB//指向线程环境块结构
……
TrapFrame//指向堆栈框架
……
ServiceTable//指向该线程关联的SSDT
……
Process//指向KPROCESS结构体
……
Win32Thread//指向W32THREAD结构体
……
ThreadListEntry//LIST_ENTRY结构体，供挂入所属进程的KTHREAD链表
……

图 4-2　KTHREAD 数据结构

"ETHREAD"中的"E"即 Executive(执行体),执行体是内核的上层结构,ETHREAD 主要供执行体层使用,例如内存管理、I/O 分发等,自然也位于内核态,数据结构如图 4-3 所示。

ETHREAD 中包括了 LPC(本地过程调用)的相关属性,LPC 是内核支持的进程间通信机制,比 RPC(远程过程调用)或其他 TCP/IP 方式更为高效,但只能用于本地主机系统之间各个进程的通信,例如视窗报文等都是通过 LPC 机制实现的。关于 LPC 后文会进行详细介绍。

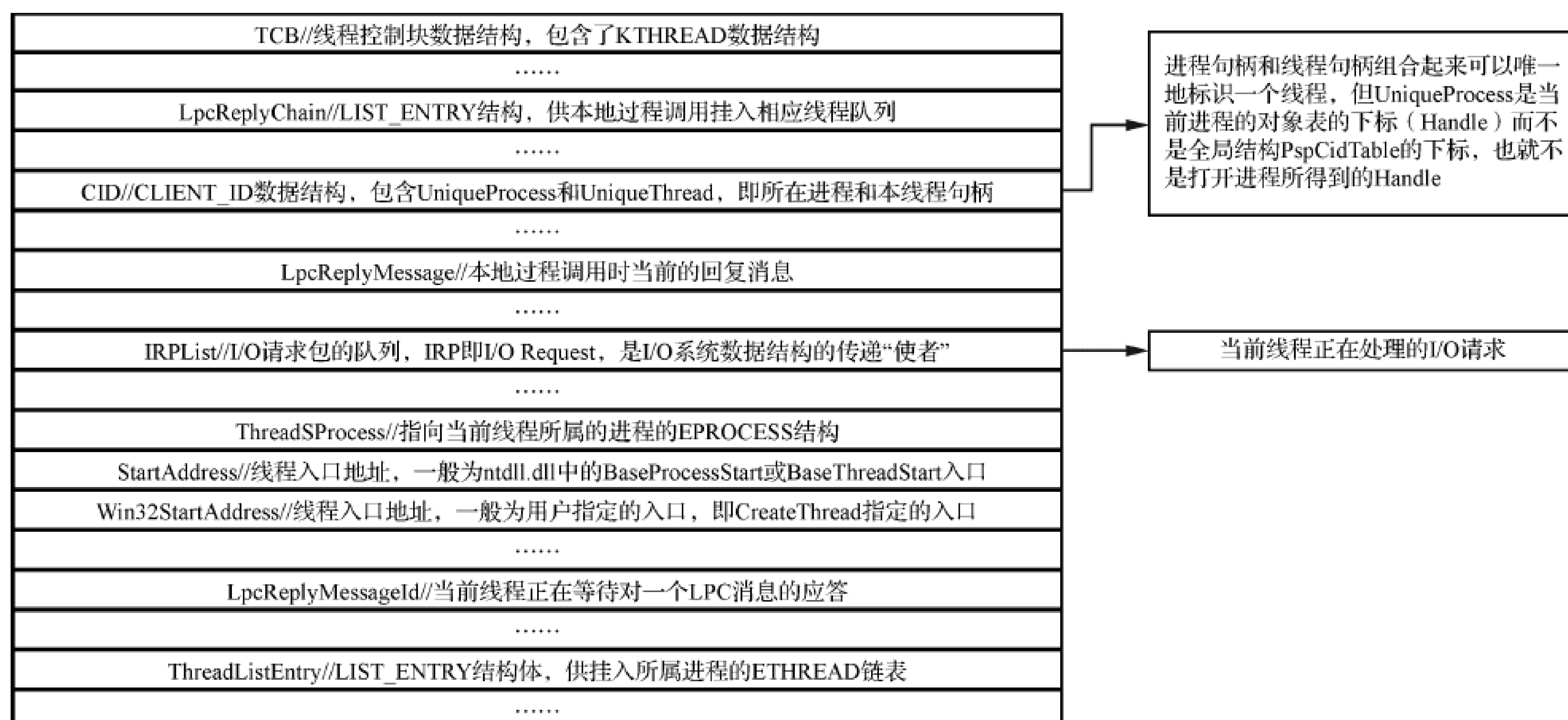


图 4－3　ETHREAD 数据结构示意图

TEB，即线程环境块，是在用户态空间使用的线程信息块。TEB 数据结构如下所示：

```
typedef struct_TEB
{
    NT_TIB NtTib;                    //NT_TIB 数据结构，里面包含了结构化异常处理例程链表
    PVOID EnvironmentPointer;
    CLIENT_ID ClientId;              //包含 UniqueProcess 和 UniqueThread，即所在进程和本线
                                     //程的句柄
    PVOID ActiveRpcHandle;
    PVOID ThreadLocalStoragePointer;
    PPEB ProcessEnvironmentBlock;    //指向所属进程的 PEB 数据结构
    ULONG LastErrorValue;
    ULONG CountOfOwnedCriticalSections;
    PVOID CsrClientThread;
    PVOID Win32ThreadInfo;           //指向 THREADINFO 结构体，里面维护了 GUI 窗口线程的各种
                                     //信息
……
} TEB, * PTEB;
```

W32THREAD 是 GUI（图形用户界面）线程所需的数据结构，承载了窗口界面显示的相关数据，供 win32k. sys 使用，自然也处于内核态。该数据结构在线程第一次调用 Windows USER 函数或 GUI 函数时创建，其结构比较小，每个 GUI 线程的 W32THREAD 结构均保存在 win32k. sys 中。在介绍视窗报文时，我们还会用到 W32THREAD 结构。W32THREAD 数据结构如下所示：

```
typedef struct_W32THREAD
{
    PETHREAD pEThread;
    ULONG RefCount;
    PTL ptlW32;
    PVOID pgdiDcattr;
    PVOID pgdiBrushAttr;
    PVOID pUMPDObjs;
    PVOID pUMPDHeap;
    DWORD dwEngAcquireCount;
    PVOID pSemTable;
    PVOID pUMPDObj;
} W32THREAD, * PW32THREAD;
```

4.1.2　进程相关数据结构

接下来介绍进程相关的数据结构。与线程数据结构对应，进程在内核态的核心层也有一个数据结构对应，称为 **KPROCESS**，以下是 KPROCESS 数据结构：

```
typedef struct _KPROCESS
{
    DISPATCHER_HEADER Header;          //与 KTHREAD 类似,这也是个分发器对象,可挂入等待队列
    LIST_ENTRY ProfileListHead;        //进程参与性能分析时,作为节点挂入全局性能分析进程列表
    ULONG DirectoryTableBase;          //指向进程页目录地址
    ULONG Unused0;
    KGDTENTRY LdtDescriptor;           //指向本进程的 LDT
    KIDTENTRY Int21Descriptor;         //DOS 环境下使用
    WORD IopmOffset;                   //本进程的 I/O 控制位图
    UCHAR Iopl;                        //I/O 优先级
    UCHAR Unused;
    ULONG ActiveProcessors;            //当前进程正在哪些处理器上运行
    ULONG KernelTime;                  //进程在内核态运行的总时间
    ULONG UserTime;                    //进程在用户态运行的总时间
    LIST_ENTRY ReadyListHead;          //链表表头,表示进程当前已经就绪但尚未加入全局就绪链表的
                                       //线程
    SINGLE_LIST_ENTRY SwapListEntry;   //当被换入内存时,通过此结构挂入全局相关链表中
    PVOID VdmTrapcHandler;
    LIST_ENTRY ThreadListHead;         //当前进程所属全部线程
    ULONG ProcessLock;
    ULONG Affinity;                    //表示本进程的 CPU 亲和性
    ……
    ULONG StackCount;                  //记录了当前进程中有多少个线程的栈位于内存中
    LIST_ENTRY ProcessListEntry;
    UINT64 CycleTime;
} KPROCESS, * PKPROCESS;
```

同样，在执行体层也对应有一个类似于 ETHREAD 的进程数据结构 **EPROCESS**，我们来看一下其具体内容。由于 EPROCESS 结构很大，在此只列出其一些比较有代表性的属性，如下所示：

```
typedef struct _EPROCESS
{
    KPROCESS Pcb;                      //EPROCESS 的第一项就是 KPROCESS 结构体,即两者地址相同
    EX_PUSH_LOCK ProcessLock;
    LARGE_INTEGER CreateTime;          //进程的创建时间
    LARGE_INTEGER ExitTime;            //进程的退出时间
    EX_RUNDOWN_REF RundownProtect;
    PVOID UniqueProcessId;             //进程 ID,即进程管理器中的 PID
    LIST_ENTRY ActiveProcessLinks;     //Windows 中,所有活动进程通过该域连成一个链表
    ULONG QuotaUsage[3];
    ULONG QuotaPeak[3];
    ULONG CommitCharge;
    ULONG PeakVirtualSize;
    ULONG VirtualSize;
    LIST_ENTRY SessionProcessLinks;
    PVOID DebugPort;                   //指向调试端口的句柄
    union
    {
        PVOID ExceptionPortData;       //指向异常端口的句柄
        ULONG ExceptionPortValue;
        ULONG ExceptionPortState:3;
    };
    PHANDLE_TABLE ObjectTable;         //指向当前进程的句柄表
    EX_FAST_REF Token;                 //该进程的访问令牌,用于该进程的安全访问检查
    ULONG WorkingSetPage;              //指向包含进程工作集(正在使用的物理页面的集合)的页面
```

```
    EX_PUSH_LOCK AddressCreationLock;
    PETHREAD RotateInProgress;
    PETHREAD ForkInProgress;
    ULONG HardwareTrigger;
    PMM_AVL_TABLE PhysicalVadRoot;
    PVOID CloneRoot;
    ULONG NumberOfPrivatePages;
    ULONG NumberOfLockedPages;
    PVOID Win32Process;             //如果不为空,则说明这是一个 GUI 进程
    PEJOB Job;
    PVOID SectionObject;            //指向内存区对象
    ……
    UCHAR ImageFileName[16];        //进程映像名称
    LIST_ENTRY JobLinks;
    PVOID LockedPagesList;
    LIST_ENTRY ThreadListHead;      //该进程中的所有线程,即 ETHREAD 结构
    PVOID SecurityPort;
    PVOID PaeTop;
    ULONG ActiveThreads;            //记录了当前进程的活动线程
    ULONG ImagePathHash;
    ULONG DefaultHardErrorProcessing;
    LONG LastThreadExitStatus;
    PPEB Peb;                        //指向进程环境块
    ……
    ULONG Flags2;                    //该域包含了进程的标志位,这些标志位反映了进程的当前状态
                                     //和配置
    ……
} EPROCESS, * PEPROCESS;
```

与 TEB 一样,进程在用户态也对应了一个 **PEB** 结构。PEB 是个比较大的数据结构,我们只选取其中一部分进行介绍,如下所示:

```
typedef struct _PEB
{
    UCHAR InheritedAddressSpace;
    UCHAR ReadImageFileExecOptions;
    UCHAR BeingDebugged;             //判断进程是否被调试
    UCHAR BitField;
    ULONG ImageUsesLargePages:1;
    ULONG IsProtectedProcess:1;
    ULONG IsLegacyProcess:1;
    ULONG IsImageDynamicallyRelocated:1;
    ULONG SpareBits:4;
    PVOID Mutant;
    PVOID ImageBaseAddress;          //进程映像基址,EXE 默认为 0x00400000,DLL 默认为
                                     //0x10000000
    PPEB_LDR_DATA Ldr;               //该结构包含了三个队列,用来记录该进程加载的模块
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;    //进程参数块
    PVOID SubSystemData;
    PVOID ProcessHeap;               //指向的是进程堆(默认的那个)的首地址
    PRTL_CRITICAL_SECTION FastPebLock;
    PVOID AtlThunkSListPtr;
    PVOID IFEOKey;
    ULONG CrossProcessFlags;
    ULONG ProcessInJob:1;
    ULONG ProcessInitializing:1;
    ULONG ReservedBits0:30;
    union
    {
        PVOID KernelCallbackTable;   //指向从内核态回调到用户态的函数指针表
        PVOID UserSharedInfoPtr;
    };
    ……
    ULONG NumberOfHeaps;             //进程堆的数量
    ULONG MaximumNumberOfHeaps;
```

```
    VOID * * ProcessHeaps;                //记录了每一个堆地址的数组
    ......
} PEB, * PPEB;
```

一个进程中的线程一旦调用了 GUI 接口，这个线程就变成了 GUI 线程，win32k. sys 也会为这个线程分配 W32THREAD 数据结构并存储于 win32k. sys。同样，如果进程中有若干个线程变成了 GUI 线程，则这个进程也会相应成为 GUI 进程，win32k. sys 会为该进程分配一个 **W32PROCESS** 数据结构，也存储于 win32k. sys 中。W32PROCESS 数据结构的各字段如下所示：

```
typedef struct_W32PROCESS
{
    PEPROCESS              peProcess;
    DWORD                  RefCount;
    ULONG                  W32PF_flags;
    PKEVENT                InputIdleEvent;
    DWORD                  StartCursorHideTime;
    struct_W32PROCESS *    NextStart;
    PVOID                  pDCAttrList;
    PVOID                  pBrushAttrList;
    DWORD                  W32Pid;
    LONG                   GDIHandleCount;
    LONG                   UserHandleCount;
    PEX_PUSH_LOCK          GDIPushLock;  /* Locking Process during access to structure. */
    RTL_AVL_TABLE          GDIEngUserMemAllocTable;  /* Process AVL Table. */
    LIST_ENTRY             GDIDcAttrFreeList;
    LIST_ENTRY             GDIBrushAttrFreeList;
} W32PROCESS, * PW32PROCESS;
```

4.2 线程创建过程

研究线程的创建过程，必须首先清楚进程的创建过程。因为进程是线程的容器，为线程运行提供各种环境和资源，无论是本地线程还是远程线程，都无法离开进程而独立存在。

进程创建过程分为以下几个步骤：

① 打开进程的目标映像文件，建立内存映射区。对于 Windows 进程，这个映像文件就是我们常说的可执行(EXE)文件，作为 PE 文件(可移植执行文件)的一种，要将其映射到内存中。

② 执行系统调用函数 NtCreateProcess 来创建进程，通过该函数初始化有关数据结构和内存地址空间，为线程的创建和运行提供环境。

③ 初始化有关数据结构和堆栈，并执行系统调用接口 NtCreateThread 来创建进程中的第一个线程(主线程)，创建好后主线程挂起并未启动执行。

④ 向环境子系统 csrss. exe(除了管理控制台窗口操作外，对进程和线程的创建与终结也需要向 csrss. exe“汇报”)通知“进程创建成功”的消息。

⑤ 启动步骤③中创建的本进程首个线程——主线程。

⑥ 执行 APC：用户空间初始化，加载和连接除系统 DLL(ntdll. dll 和 kernel32. dll)外的各个 DLL。

步骤①非常简单,限于篇幅我们在此略过,重点来看步骤②、③和⑤,这几步是理解线程和进程创建的核心。先来看步骤②。

1. 步骤②

1) 创建 EPROCESS

首先需要创建 EPROCESS 结构体,这是进程在执行体层的体现和操作抓手,不但要创建还要初始化。这里要强调的是 EPROCESS 是被创建进程的数据结构而不是当前进程(创建者进程)的,当前进程作为被创建进程的父进程(例如 explorer. exe)为其分配数据结构,父进程自己的数据结构早就完成了,我们在此以“子进程”指代被创建进程。

2) 创建句柄表和进程内存页面表

接下来要为子进程创建句柄表和进程内存页面表(后文在内存管理相关章节中会有详细描述),同时为页面表复制一份系统空间的页面映射关系。因为在 Windows 系统中,所有进程的地址空间的系统空间部分(例如 32 位系统中地址空间高于 0x80000000 的部分)的地址映射是一样的(严格来说也有几处不一样,后文详细描述,这里忽略不计)。

3) 初始化 KPROCESS

创建完页面映射关系后,NtCreateProcess 开始初始化 KPROCESS 结构。注意,这里 KPROCESS 已经创建过了,可以直接初始化。因为 KPROCESS 作为进程控制块(PCB)是 EPROCESS 的一部分,是 EPROCESS 中的第一个数据结构,创建 EPROCESS 时自然也就同时分配了 KPROCESS 的内存,相当于创建了 KPROCESS,只是没有初始化。

4) 创建用户态内存空间

接下来要创建子进程的用户态内存空间,并将步骤①中的 EXE 内存映射区再映射到子进程地址空间中。但我们当前处于父进程中,怎么在父进程中将这个内存映射区映射到子进程地址空间中呢?

Windows 提供了“进程挂靠”机制,也就是说在创建用户态的数据结构前,将当前线程(父进程中的线程)挂靠到子进程中,就像将它“过继”给别人一样,等完成了一系列创建工作再把它“归还”回来,如此,创建过程就像是在当前进程自己的空间中进行一样。

在此要调用 KeAttachProcess 将当前线程(父进程中正在执行子进程创建的线程)“过继”给子进程后再执行,这样创建的地址映射表就是子进程的了。上述工作完成后再调用 KeDetachProcess 将父进程与子进程脱离,父进程“回归”,继续之前的“事业”。

在 32 位系统中每个进程的用户态空间和内核态空间的大小之比大致为 1 : 1 或者 3 : 1。3 : 1 的情况比较少见,这需要更改配置,大多数情况下还是 2 GB 的用户态空间和 2 GB 的内核态空间(1 : 1 格局)。但是这 2 GB 内存空间也不是都能使用,比如在内存空间中上、下限的 64 KB 内存空间就是禁区,是无法访问和使用的。如图 4 - 4 所示是 NtCreateProcess 执行完成后用户态空间的内存布局。

注:这里所说的内存空间是虚拟内存空间而不是物理内存空间,虚拟内存空间的大小和初始布局对每个进程都是一样的,关于虚拟内存空间和物理内存空间后文会有详细描述。

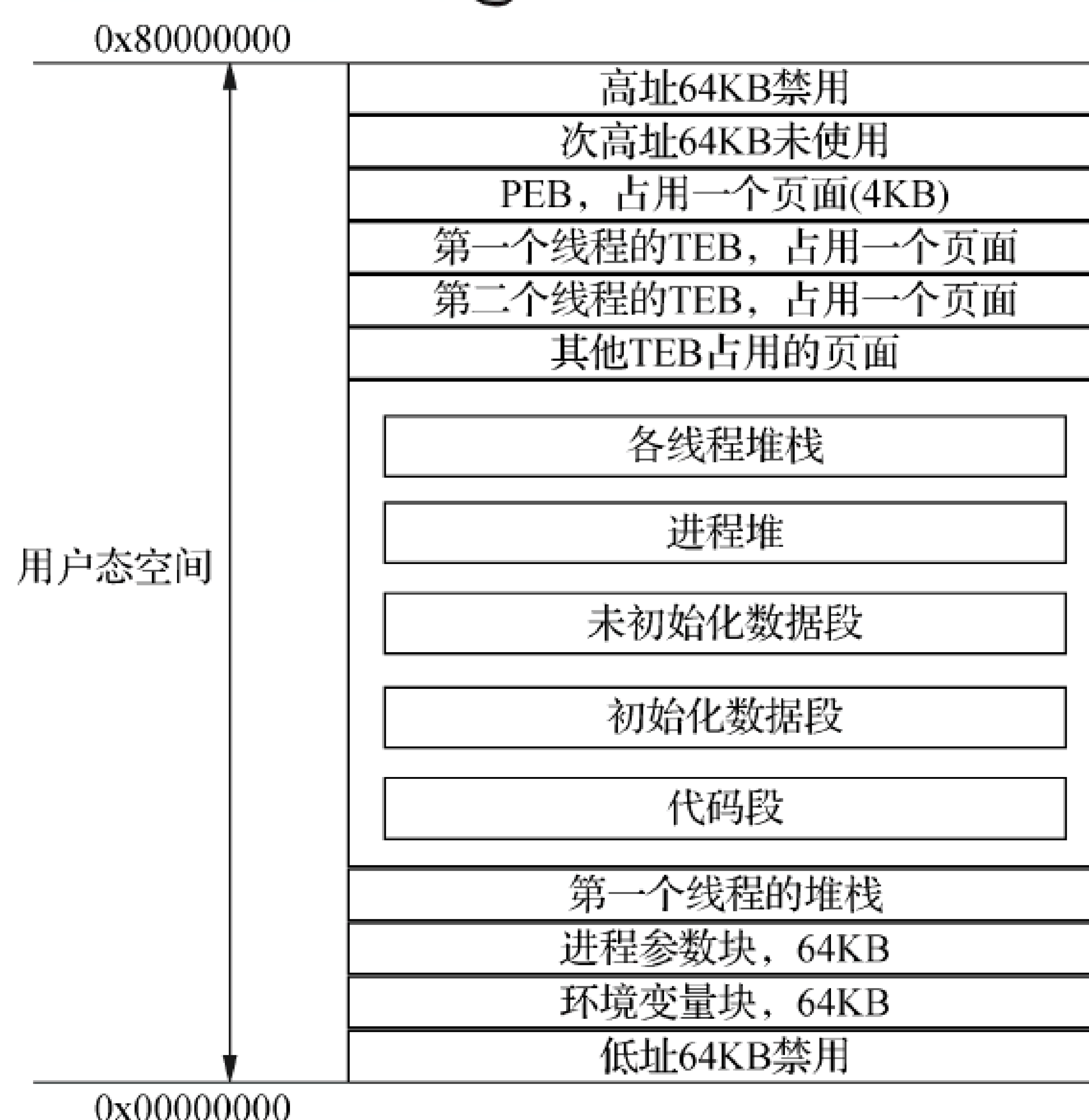


图 4－4　用户态内存空间示意图

5）映射 ntdll. dll

用户态内存空间布局完成后还要把 ntdll. dll 映射到内存空间中。ntdll. dll 是 Windows 系统中非常重要的系统库，许多系统调用接口函数、APC 分发机制相关函数、进程/线程初始入口函数、DLL 加载连接机制函数等关键方法都“硬写”在 ntdll. dll 中。也就是说，ntdll. dll 要作为系统中第二个被加载的 PE 模块紧随在 EXE 文件之后。

因为进程创建过程执行到这里还有许多必要的模块没有被加载。我们知道一个进程要执行，其 EXE 文件除了自身的一些固有功能外大多还要依赖与其他 PE 模块的“赋能”。可是这些模块怎样为 EXE 赋能呢？而这些基本都运行在用户态空间的模块怎么才能在内核态进行加载和连接呢？这里面就涉及异步过程调用（APC）机制了，APC 在从内核态返回用户态的空当执行，而这个“空窗时机”正好是各个模块进行加载和连接的最好时机，因为一旦回到用户态就要执行入口函数了，而这要求所有模块必须已经连接。

模块的连接要用到 PE 导入表、导出表机制，我们在后文中会分别介绍 APC 和 PE 模块的这些机制。

6）创建和初始化 PEB

接下来要创建和初始化进程环境块（PEB）。PEB 是进程在用户态的体现，以方便执行用户态进程的各种操作。

但我们知道，在虚拟地址空间中，各进程的内核态地址空间映射是基本一致的，但是在用户态却各有各的空间映射。PEB 是用户态的数据结构，虽然其创建的基址都一样，但这毕竟是子进程的数据结构，而我们当前处于父进程上下文中，创建的 PEB 要体现在子进程的地址映射中，怎么办？方法还是借用“进程挂靠”机制，PEB 也好，TEB 也罢，都会体现在子进程的地址映射中。有关进程挂靠的机制后文还会描述，这里只是简单提一下。

PEB 创建完成后,便将 EPROCESS 加入到 PsActiveProcessHead 链表中。这个链表是 Windows 系统中的全局链表,记录着系统中的所有进程。

至此,进程环境构建完毕,步骤②执行完成,其过程如图 4－5 的上半部分所示,线程创建和运行的容器环境已经具备,接下来要创建子进程的运行实体——主线程。

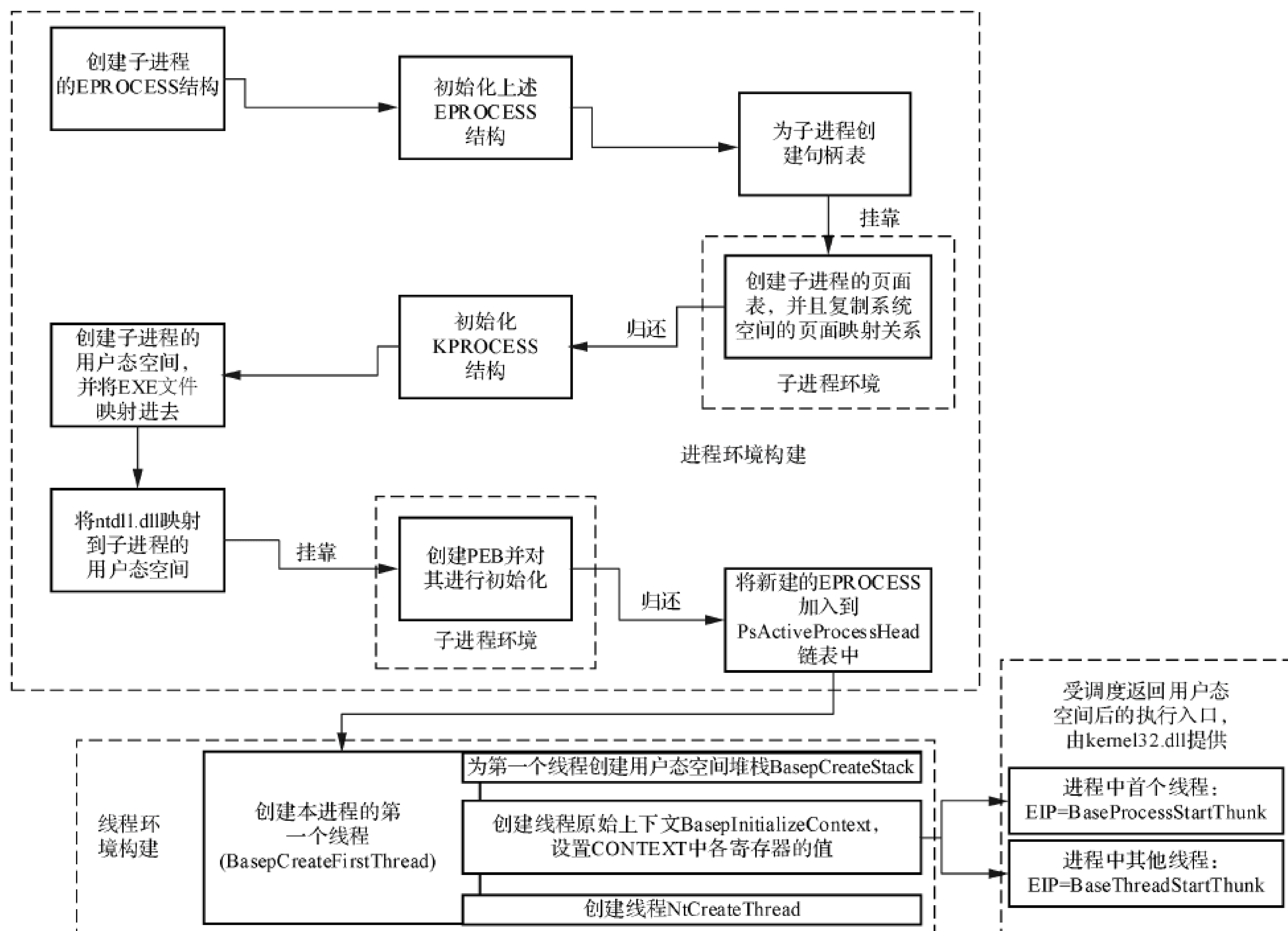


图 4－5　NtCreateProcess 执行全过程示意图

2．步骤③

如图 4－5 的下半部分所示,创建子进程的主线程有三个基本步骤:

第一步,创建主线程的用户态堆栈。我们在创建进程的时候已经完成了虚拟地址空间的创建和初始化。但由于堆栈是与线程相关的,因此只能在创建线程的时候创建堆栈。

第二步,创建原始上下文,设置上下文的各个域值。这个上下文作为参数传递给下一步的 NtCreateThread,包含了线程的入口地址、非 APC 方式返回用户态空间后的指令指针等重要信息。例如进程主线程(首个线程)返回用户态后的入口函数是 BaseProcessStartThunk,非首个线程的入口函数则是 BaseThreadStartThunk。这个上下文营造了线程在真正返回用户态执行时的上下文环境。

第三步,上述两步完成后开始线程创建的实质操作——执行 NtCreateThread。NtCreateThread 执行全过程如图 4－6 所示。

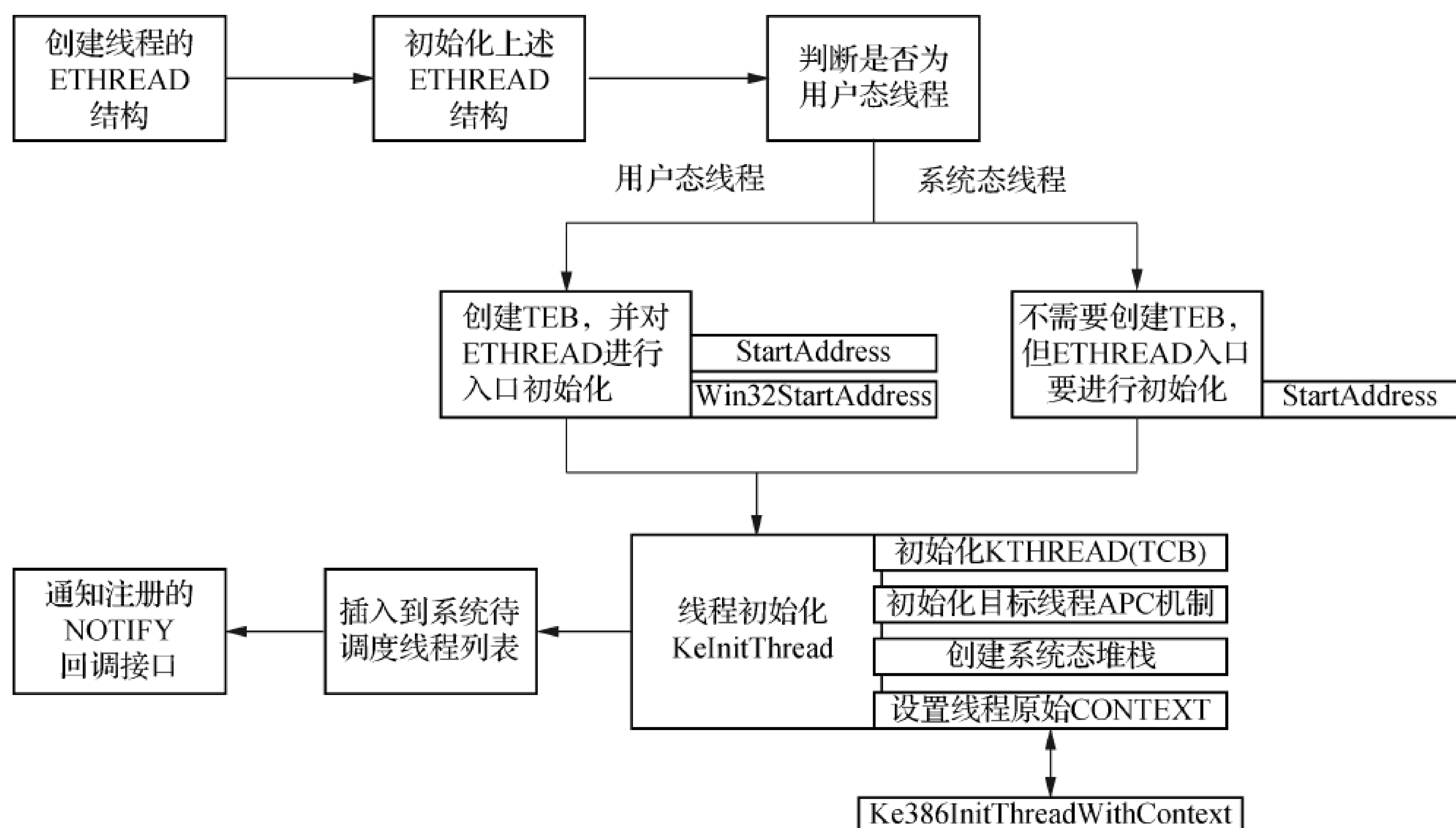


图 4－6　NtCreateThread 执行全过程示意图

1）解构 NtCreateThread 创建线程

如图 4－6 所示，我们直接从 NtCreateThread 开始看创建线程的主要步骤。

（1）创建和初始化 ETHREAD

首先创建线程的执行体对象 ETHREAD。与进程类似，线程在执行体层也具有对应的执行体层数据结构 ETHREAD，其具体域值在上文也有描述，创建完成后还需要对其初始化。

（2）为线程准备数据结构

ETHREAD 完成初始化后，要判断被创建的线程是用户态线程还是内核态线程，接下来为这两种线程准备的数据结构是不一样的。

针对用户态线程要创建和初始化 TEB。与 PEB 类似，TEB 是线程在用户态的数据结构，方便针对当前线程进行用户态的操作。

除了构建 TEB，还需要对 ETHREAD 中的线程入口赋值，比如 StartAddress 和 Win32StartAddress，前者是 kernel32. dll 为线程准备的入口，对于进程中的首个线程为 BaseProcessStartThunk，对于其他线程则为 BaseThreadStartThunk。如图 4－6 所示，这两个接口是通过线程创建过程的第二个步骤中的上下文参数 CONTEXT 传到 NtCreateThread 中的。Win32StartAddress 是由编译器或用户指定的线程入口，针对主线程（一般就是首个线程）编译器可设置为 mainCRTStartup，它为线程准备一些运行环境，特别是设置结构化异常处理机制来保护线程的执行；对于其他的线程，则由用户来指定入口 OEP（Original Entry Point，原始入口点）。

针对内核态线程就不需要创建 TEB 了，这种线程不会在用户态运行，自然也就不需要用户态的数据结构了，有了反而还是累赘，但入口函数还是要的，即 ETHREAD 的 StartAddress 域，其值与用户态的 StartAddress 域值一样，作用相同，但是 Win32StartAddress 就不需要了。

（3）初始化线程

上述两种线程殊途同归后则马上初始化线程，即执行 KeInitThread 函数，这是线程创建最重要的步骤，包括初始化 KTHREAD 结构、初始化目标线程的 APC 机制、创建线程内核态堆栈、在内核态堆栈中设置线程运行上下文等，我们稍后再来展开这些步骤。

（4）线程挂起等待调度

线程初始化完成后，会将线程插入待调度队列中，线程处于挂起状态。此时的线程就好像是刚刚执行完时间片后被剥夺了执行权一样，安安静静地等待下一次被调度的时机。还会向环境子系统注册，通知它有个线程已经创建完成了，还会通知其他注册了线程创建通知回调的模块：该线程已创建完成。至此，线程创建万事俱备，只欠调度执行的东风了。我们回头再来看 KeInitThread 对线程的初始化。

2）解构 keInitThread 初始化线程

（1）初始化 KTHREAD

与进程类似，在内核核心层线程也有对应的数据结构，即 KTHREAD，也就是线程控制块（TCB）。因为 TCB 是 ETHREAD 的第一个数据结构（注意，这里不是指针），因此为 ETHREAD 分配内存的同时也就构造好了 KTHREAD，因此这里只需要初始化。

（2）初始化线程的 APC 机制

APC 的名称很好地描述了它的运行方式。在从内核态向用户态返回的过程中，Windows 会自动执行 APC，APC 在队列中等待被执行。在线程和进程创建过程中，除了 EXE 文件和 ntdll. dll 的加载，其他 PE 文件的导入、加载和连接都要交给 APC 来完成，这是因为这些库无法在内核态完成导入、加载和连接，只能回到用户态实现。而且回到用户态还不算完，还要能再次返回内核态，就像时光穿梭机一样有来有回，这种神奇的操作我们会在后文详细描述，现在我们只要知道有这么回事就行。

（3）创建内核态堆栈

接下来要创建内核态堆栈，前面的步骤中已经针对首个线程创建了用户态堆栈，这里没有太多可解释的。

（4）构造返回用户态空间的上下文

最后要通过 Ke386InitThreadWithContext 设置线程原始上下文（CONTEXT），这个 CONTEXT 大有讲究，其作用是在内核态堆栈中构造一些“假”的框架，伪造被创建线程在用户态空间的运行环境，使其在返回用户态的过程中按照我们伪造的这个上下文来执行，例如进入伪造的入口函数等。如图 4－7 所示为“假”框架的布局，其中：

- 堆栈最底部是 FX_SAVE_AREA 框架，用于浮点处理器；
- 沿着堆栈生长方向往上一层是自陷框架 KTRAP_FRAME，这个框架与我们在系统调用中讲述的自陷框架的结构完全一致，当然也只有用户态线程的系统调用才会存在自陷框架；
- 再往上是一个函数调用框架 StartFrame，是为最上层的切换框架 CtxSwitchFrame 中的

一个接口提供参数的。

我们来看图 4－7 中最右边的虚线框，里面描述了切换框架和调用框架的实际值。

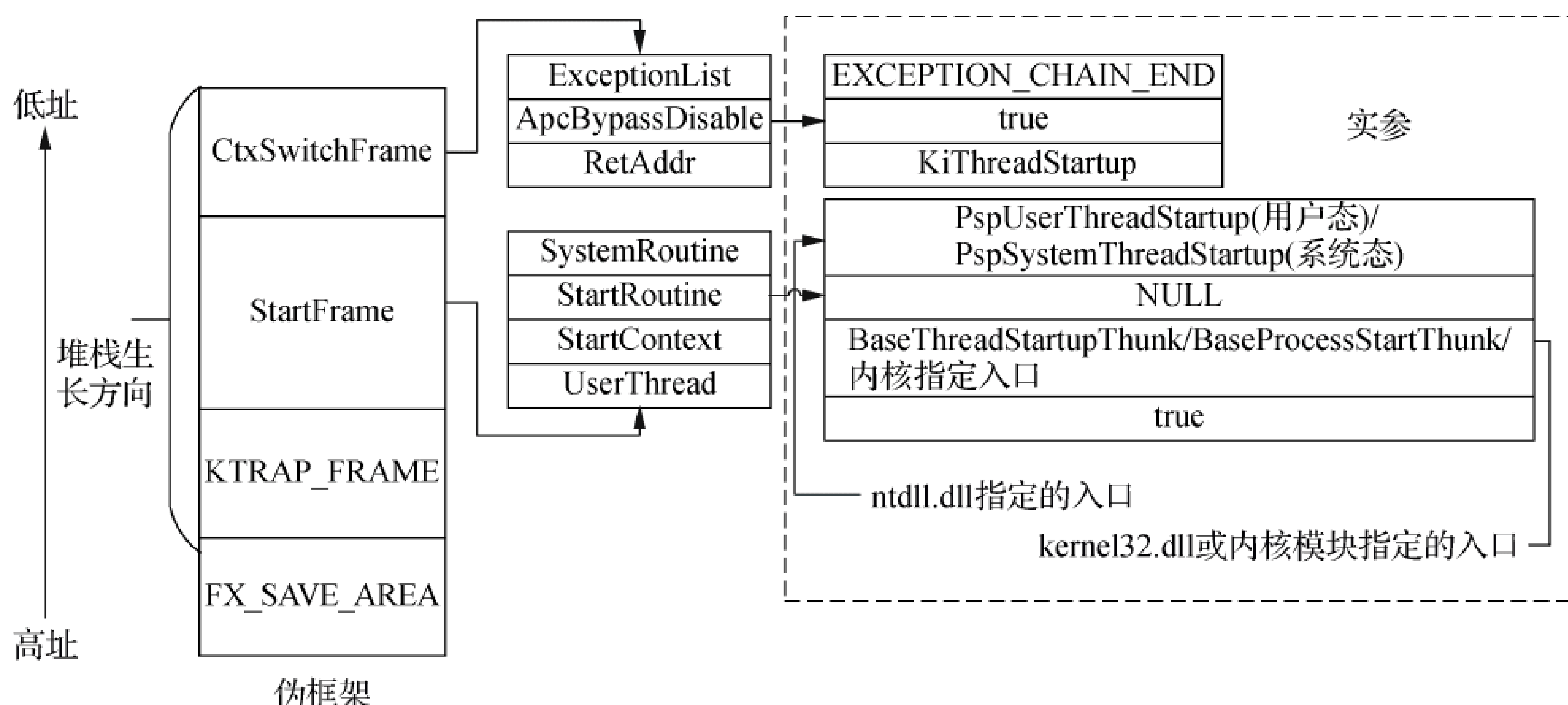


图 4－7　Ke386InitThreadWithContext 在内核态堆栈设置的伪框架

3．步骤⑤

前文说过，线程初始化完成后会挂入系统待调度队列，从线程自身的角度看，仿佛之前已经执行过了，只是用完了时间片后被剥夺了执行权，等待下一个执行周期来临时继续执行，我们就从这里展开。

1）弹出和执行返回地址指针

假如这个执行周期到来了，系统调度框架从队列中取出该线程，此时线程刚刚“出狱”，处于内核态，系统堆栈中存放的是上一步构造的伪框架。被调度时首先从 CtxSwitchFrame 框架中弹出前两项，继而执行返回地址指针 RetAddr，这个指针的实参为 KiThreadStartup 函数，也就是要执行该函数，故事的“点睛之笔”就在这里（CtxSwitchFrame 就是切换框架，对于线程首次调度，其 RetAddr 硬写为 KiThreadStartup，否则为 call KiSwapContextInternel 的下一条指令，意即执行完线程切换后的下一条汇编指令）。

2）通过 KiThreadStartup 返回用户态空间

线程首次调度时从 KiThreadStartup 开始执行，KiThreadStartup 的参数从右向左是这样的：TrapFrame、UserThread、StartContext、StartRoutine 和 SystemRoutine，有心的读者会发现这个参数布局与系统堆栈中间两个框架（图 4－7 中伪框架中的 KTRAP_FRAME 和 StartFrame）完全一致。其实这样的布局正是为 KiThreadStartup 参数准备的，使 KiThreadStartup 的参数在系统堆栈中都能出现。而 KiThreadStartup 函数的作用是：

- 中断请求级别降低到 APC_LEVEL。
- 执行 SystemRoutine，对于用户态线程即为 PspUserThreadStartup。
- 通过 KiServiceExit 返回用户态空间。返回用户态空间时，内核态堆栈中伪造的框架都被消耗掉了，堆栈配平。

3）返回用户态空间后执行总入口

返回用户态空间后，由于先前在内核态堆栈构造的 TRAP_FRAME 中的 EIP 指针为 kernel32. dll 提供的 BaseProcessStartupThunk 或 BaseThreadStartThunk，即线程的用户态空间总入口，因此返回后会从这个总入口开始执行，继而执行用户提供的总入口 OEP。

4）步骤⑤的梳理

不难发现，总入口执行的核心是 SystemRoutine。SystemRoutine 的实参是 ntdll. dll 提供的 PspUserThreadStartup（用户态线程）或 PspSystemThreadStartup（内核态线程），作为用户态的入口，PspUserThreadStartup 的作用如下：

- 将中断请求级别上升到 APC_LEVEL。
- 初始化模块加载的 APC 并入队：APC 以 ntdll. dll 中的 LdrInitilizeThunk 作为 APC 例程挂入该线程队列，该函数的作用是加载和连接其他 PE 模块，但不会立即执行，而是等待从内核态切换到用户态或中断请求级别从 APC_LEVEL 降到 PASSIVE_LEVEL 时才会执行。
- 从 APC_LEVEL 返回原中断处理级别。

可以认为 PspUserThreadStartup 的作用主要是加载和连接剩余的 PE 模块。

等 PspUserThreadStartup 执行完成，伪框架中的 CtxSwitchFrame 和 StartFrame 也都消耗完了，也就该执行用户态空间的返回操作了。返回操作都执行完成就会正式回到用户态空间，剩下的事情就与系统调用返回如出一辙了。

我们以用户态线程创建为例，直观展示一下进程和线程创建的过程，如图 4－8 所示。

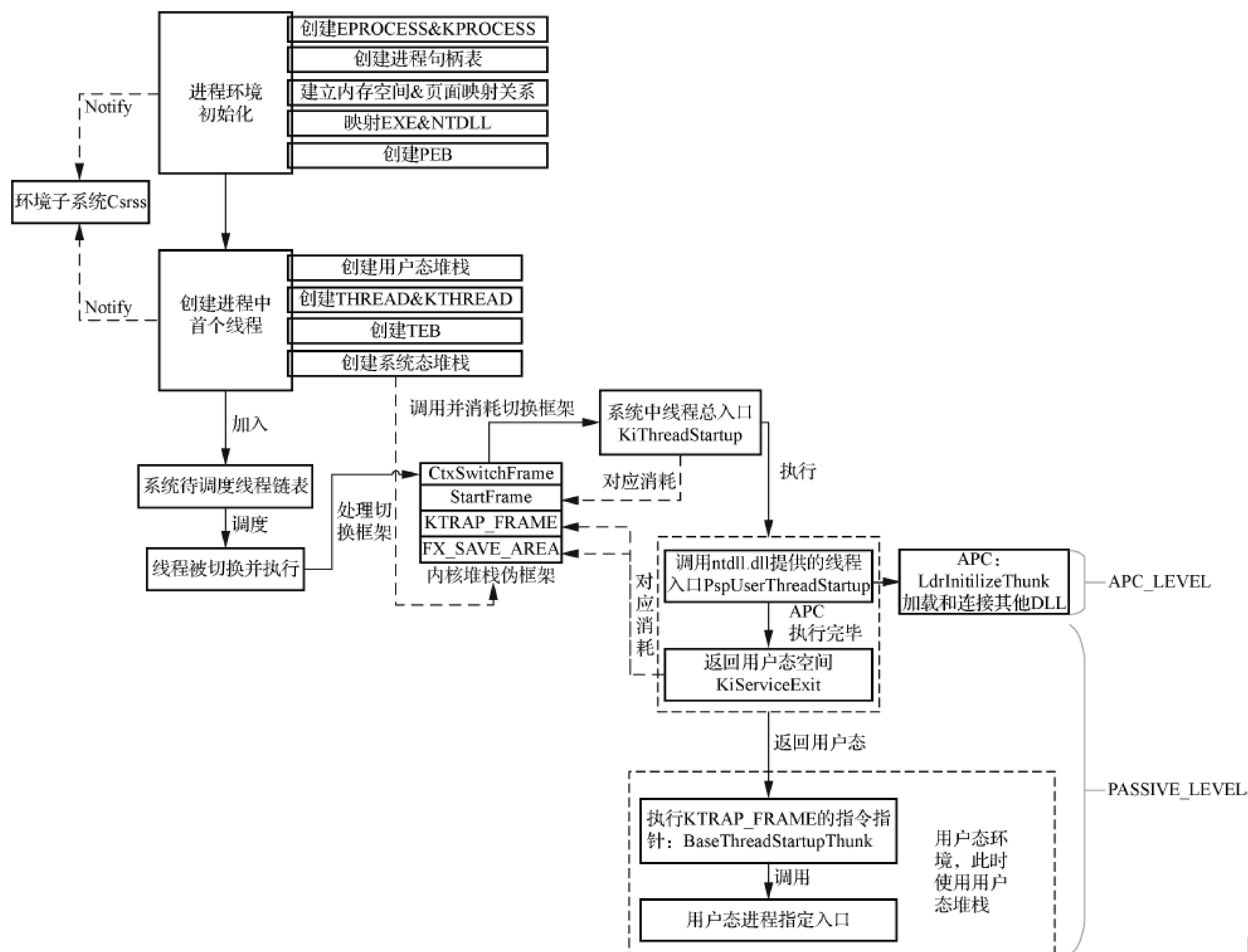


图 4－8　用户态线程创建全过程





注意，在执行 APC 的过程中，LdrInitilizeThunk 可以在用户态对其余的 PE 模块进行加载和连接，这是在返回用户态空间之前就执行的。LdrInitilizeThunk 是 ntdll. dll 中的接口，不需要连接便可直接调用。最后执行用户态线程，进程和主线程的创建工作也就完成了。

整个执行过程中，线程一共有两次回到用户态：

➢ 调度的时候出于降低中断请求级别的原因执行 APC，APC 会返回到用户态执行动态库加载函数 LdrInitilizeThunk，执行完再次返回内核态；

➢ 上一步返回内核态后触发 KiServiceExit，再次返回用户态后执行 PspUserThreadStartup，这也是一般的线程入口函数。

还要注意的是，LdrInitilizeThunk 会创建进程的堆结构，以存储加载和连接的 PE 模块信息。堆与栈不同，堆的生长方向是从低址向高址。具体地看，LdrInitilizeThunk 创建完堆以后会分配一个 PEB_LDR_DATA 结构，该结构中包含了三个链表，用于记录所加载的模块，这三个链表通过 LDR_DATA_TABLE_ENTRY 结构将加载的模块串联起来：

➢ InLoadOrderModuleList：表示模块按加载顺序挂载到链表中。

➢ InMemoryOrderModuleList：表示模块按内存基址由低到高的顺序挂载到链表中

➢ InInitializationOrderModuleList：表示模块按初始化的先后顺序挂载到链表中。

在 PEB_LDR_DATA 结构之上，再为 ntdll. dll 和 EXE 文件分配 LDR_DATA_TABLE_ENTRY 数据结构，记录模块基址、入口等信息，并且今后每加载一个模块就在堆中按堆生长的方向分配一个 LDR_DATA_TABLE_ENTRY 结构，如图 4－9 所示。当然，模块和模块之间还会存在依赖关系，在加载一个新的模块时，若它依赖另外的一个模块，会先到 InLoadOrderModuleList 搜索一下，如果有则复用，没有则新建。

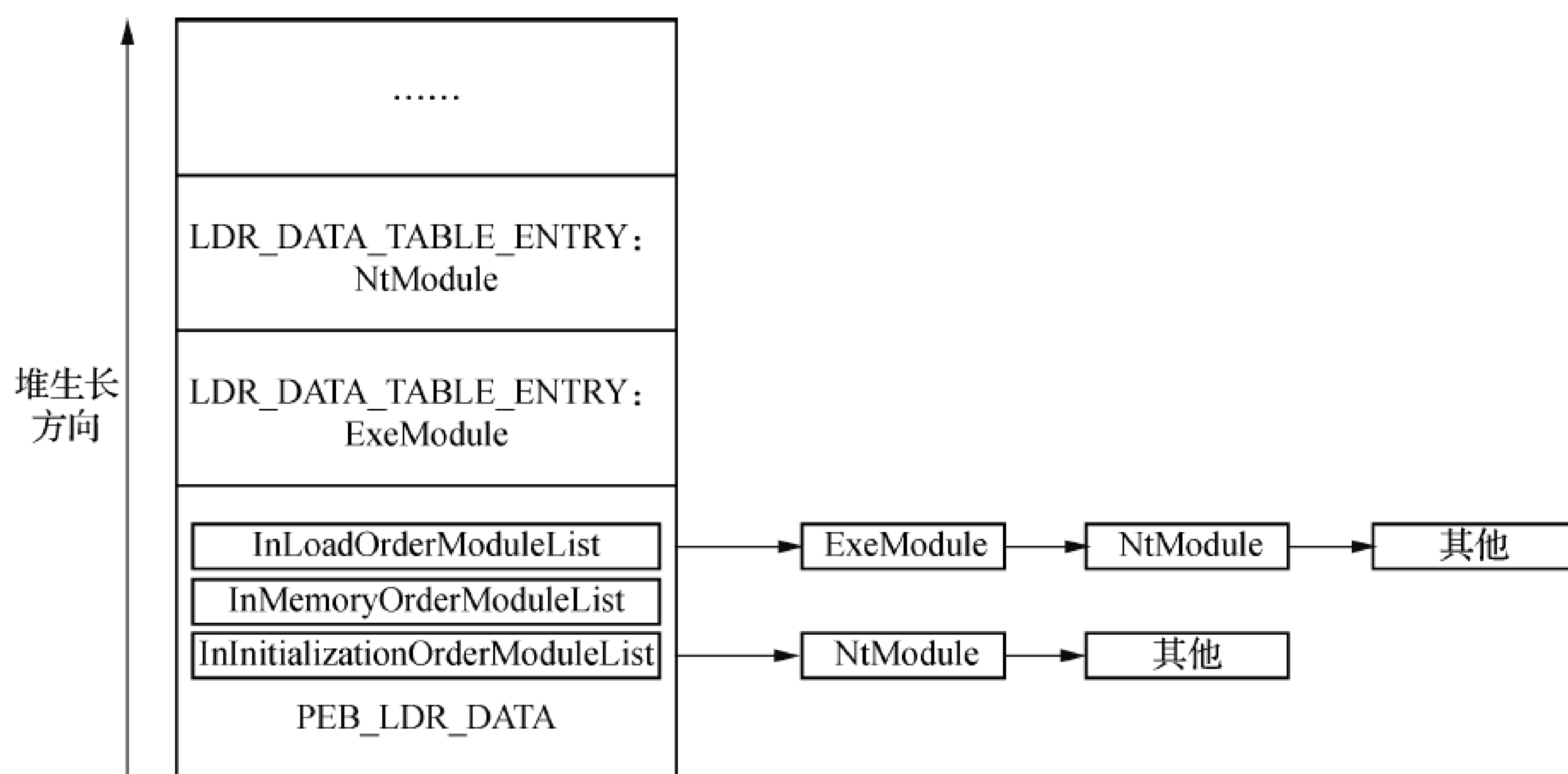


图 4－9　LdrInitilizeThunk 为模块加载分配的数据结构

至于其他具体的加载和连接细节，包括 PE 导入表、绑定导入、导出表、重定位等，由于涉及 Windows PE 文件的相关技术并且比较繁琐，我们在这里暂时不做展开。

本章小结

线程是操作系统运行的基本单位。在 Windows 中之所以能运行远多于 CPU 核数的进程数完全有赖于线程调度机制。本章介绍了线程和进程的数据结构以及线程创建的过程。线程的创建过程本质上也是两个状态(用户态和内核态)的堆栈“谋篇布局”的过程。

第 5 章　线程调度与切换

我们知道,在操作系统中进程运行的基本单位是线程。每个线程都会在 CPU 上执行一段时间(CPU 赋予的时间片),这个时间片可能是一个或几个系统时钟周期,而时钟周期是非常短的。我们在使用操作系统的时候,即使开了很多进程也可以比较流畅地使用,一方面是因为每个进程中有很多线程是不运行的(例如线程处于等待状态而不在被调度执行的范畴),但主要还是因为线程调度与切换机制,利用时间片快速地切换线程,让每个线程"雨露均沾"CPU 时间,从而造成一种假象:系统中进程的每个线程都在运行。在 CPU 数量有限而线程数量无限的情况下,就是靠这种"三个锅两个盖来回倒腾"的方式保证了进程的实时性和可交互性,这也是现代操作系统的基本特征。

本章我们将按照图 5－1 所示的提纲来介绍线程调度与切换是怎么回事。

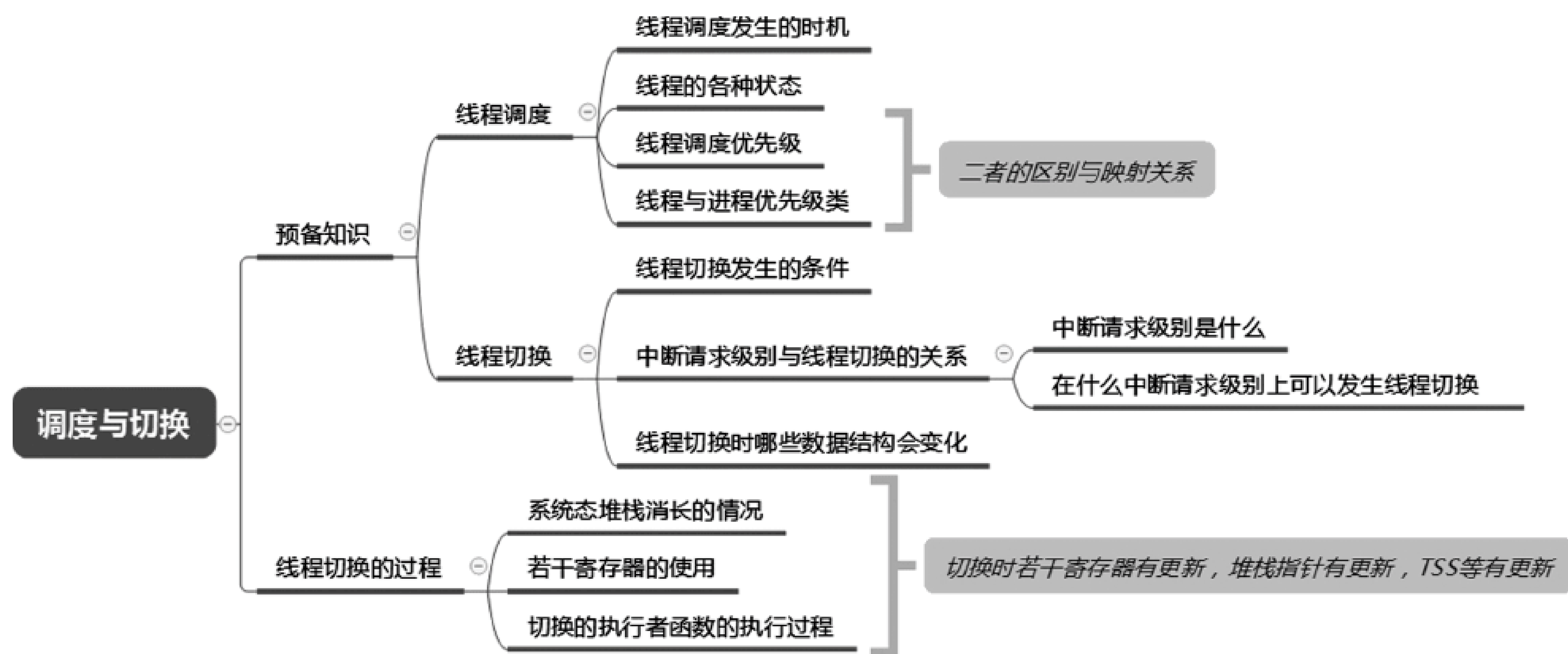


图 5－1　本章提纲

5.1　预备知识

5.1.1　线程调度

线程调度就是保证线程按顺序执行,分享 CPU 时间片的机制。线程调度发生的时机包括以下几类:

- 当前线程通过系统调用 NtYieldExecution 自愿礼让。
- 当前线程在别的系统调用中因操作受阻而半自愿地交出运行权。
- 当前线程通过系统调用 NtSetInformationThread 等改变了自身或其他线程/进程的优先

级,使得自己不再具有最高的运行优先级从而让出运行权。

- 当前线程通过系统调用 NtSuspendThread 挂起自身的运行。
- 当前线程通过系统调用 NtResumeThread 恢复了其他线程的运行,可能会使自己不再具有最高的运行优先级从而让出运行权。
- 当前线程通过进程间通信、线程间通信唤醒了别的进程,使得自己不再具有最高的运行优先级。
- 当前线程的时间片用完,因而调度其他线程运行。
- 中断的发生导致某个或某些线程被唤醒,从而使得当前线程不再具有最高的运行优先级从而让出运行权。

在系统中最常见的就是时间片用完而调度其他线程运行。

在此首先介绍下线程状态的概念。线程状态包括运行状态、就绪状态、等待状态、消亡状态。

- **运行状态**:线程正在执行的状态(分享 CPU 赋予的时间片)。
- **就绪状态**:线程在队列中等候,即将被调度到 CPU 上运行时的状态。在 KPRCB 中有个 DispatcherReadyListHead 队列数组,共有 32 个元素,代表了 32 个线程调度优先级(0—31),每个元素代表一个相应优先级的线程队列,线程被 push 到对应的队列中等待执行。执行是按照优先级从高到低的顺序执行的(31 优先级最高)。
- **等待状态**:线程由于等待系统中的某个资源(例如锁)或调用休眠指令(Sleep)而在等待的状态。这时的线程不在就绪队列中,只有等到了想要的资源才会被调度到对应的就绪队列。
- **消亡状态**:线程执行完毕或者被人为终止,或者遭遇异常使线程无法继续执行而终止的状态。

线程的切换与调度是两个维度的概念。所谓线程切换,就是两个线程执行状态的转变,涉及系统堆栈、指令指针寄存器以及其他寄存器上下文的转换,而且切换的过程中不允许 PASSIVE_LEVEL 和 APC_LEVEL 级别中断的发生。而线程调度是在当前线程的执行时间片使用完成后决定下一个时间片让哪个线程来执行,或者更高调度优先级的线程就绪的时候决定是否抢占当前线程的执行时间片。线程切换是线程调度的结果,但是线程调度的结果却不一定是线程切换。在此要体会两者的不同。

上文提到了线程一共有 32 个线程**调度优先级**,但实际上 Windows 支持的线程**优先级类**只有 7 个,包括 idle、lowest、below normal、normal、above normal、highest 和 time-critical,其中我们比较常见的是 normal 和 idle。Windows 还定义了进程优先级类,包括以下 6 个:idle、below normal、normal、above normal、high 和 real-time,比如任务管理器进程就运行在 high 级别,因此无论系统有多少进程在运行,调用任务管理器时系统总会立即响应。

这里就比较令人困惑了:又是进程优先级类,又是线程优先级类,那它们和 32 个线程调度优先级有什么关系?它们是怎么反映到线程调度的具体操作中的?其实,Windows 将上

述进程/线程优先级类按照矩阵对应的方式与 32 个调度优先级进行了映射,如表 5－1 所示,表中的数字表示线程调度优先级。

表 5－1　进程/线程优先级类与线程调度优先级的映射关系

线程优先级类	进程优先级类					
	idle	below normal	normal	above normal	high	real-time
time-critical	15	15	15	15	15	31
highest	6	8	10	12	15	26
above normal	5	7	9	11	14	25
normal	4	6	8	10	13	24
below normal	3	5	7	9	12	23
lowest	2	4	6	8	11	22
idle	1	1	1	1	1	16

应用程序不需要自己设置**调度优先级**,但可以在创建进程和线程的时候指定**优先级类**,当线程被调度的时候会按照上述映射关系进行调度优先级设置。线程在执行时间片中优先级还会略微下降,“占用了时间,降低了级别”,这种公平机制也是比较合乎情理的。

5.1.2　线程切换

下面我们来看一下线程切换,线程切换需要一定的条件。

首先,线程的切换只能发生在内核态(R0),也就是说只有线程进入内核态才有可能进行切换。当然,现在也有一些手段可以在用户态切换线程,例如 DPDK 等一些高 I/O 通量的通信框架,从协议栈到 I/O 管理,再到整个框架的线程调度和线程切换都在用户态(R3)空间运行,但这不属于操作系统线程调度和线程切换的范畴,我们在此不做描述。

其次,线程切换需要 CPU 处于一定的中断请求级别(Interrupt Request Level,IRQL)。所谓中断请求级别其实是 Windows 提出的逻辑概念,就是划分了在系统中的中断优先级,CPU 运行在这些中断优先级上,以决定中断(硬中断＋软中断)是否能打断当前线程的执行。定义了优先级,有些系统行为(例如线程切换)就只能在规定的级别发生,还有些系统行为(例如中断响应)可以抢占优先级更低的线程的执行权。

Windows 定义的中断请求级别如表 5－2 和 5－3 所示。

表 5－2　X86 架构下的中断请求级别定义

优先级	优先级定义	优先级说明	中断类型
0	PASSIVE_LEVEL	普通线程执行级别	软中断
1	APC_LEVEL	异步过程调用(APC)	软中断
2	DISPATCH_LEVEL	线程调度、延迟过程调用(DPC)	软中断
3	DIRQL	设备中断、硬件中断,由设备驱动定义	硬中断

续表 5－2

优先级	优先级定义	优先级说明	中断类型
……	……	设备中断、硬件中断,由设备驱动定义	硬中断
26	DIRQL	设备中断、硬件中断,由设备驱动定义	硬中断
27	PROFILE_LEVEL	性能分析级别	硬中断
28	CLOCK_LEVEL	时钟中断级别	硬中断
29	IPI_LEVEL	处理器间中断协同级别	硬中断
30	POWER_LEVEL	电源级别	硬中断
31	HIGH_LEVEL	高	硬中断

表 5－3　X64 架构下的中断请求级别定义

优先级	优先级定义	优先级说明	中断类型
0	PASSIVE_LEVEL	普通线程执行级别	软中断
1	APC_LEVEL	异步过程调用(APC)	软中断
2	DISPATCH_LEVEL	线程调度、延迟过程调用(DPC)	软中断
3	CMC_LEVEL	可矫正机器检查级别	硬中断
4	DIRQL	设备中断、硬件中断,设备驱动定义	硬中断
……	……	设备中断、硬件中断,设备驱动定义	硬中断
11	DIRQL	设备中断、硬件中断,设备驱动定义	硬中断
12	PC_LEVEL	性能计数器级别	硬中断
13	SYNCH_LEVEL/CLOCK_LEVEL	同步级别/时钟中断级别	硬中断
14	IPI_LEVEL	处理器间中断协同级别	硬中断
15	PROFILE_LEVEL	性能分析级别	硬中断

中断请求级别从 0 到 31 依次升高,也就是说 Windows 一共定义了 32 个 IRQL。中断请求级别要与线程调度优先级区分开,二者完全是两回事,虽然都分为 32 个级别,但中断请求级别是对 CPU 说的,而线程调度优先级则是对线程说的。

线程调度优先级无论有多少级,都只能处在 IRQL 的 DISPATCH_LEVEL 优先级以下,因为 DISPATCH_LEVEL 是线程调度发生的级别,无论线程有什么样的优先级,都只能在这个级别上发生调度,就好像是"线程优先级这个孙悟空再怎么七十二变也跳不出 DISPATCH_LEVEL 这个如来的手掌心"。

在此我们展开讨论一下 IRQL,首先来看软中断。顾名思义,软中断就是由软件产生的中断,无论在 X86 架构下还是在 X64 架构下软中断都有三个优先级:PASSIVE_LEVEL(0)、APC_LEVEL(1)和 DISPATCH_LEVEL(2)。

➢ **PASSIVE_LEVEL**:最低优先级的 IRQL,其他 31 个 IRQL 中的任何一个都可以中断





它。普通线程(非实时的、非 APC 和 DPC 的线程)一般运行在这个 IRQL。这里要强调一点,线程运行时 IRQL 不是一成不变的,就拿普通线程来说,在进入线程切换阶段时 CPU 要经历从 PASSIVE_LEVEL 到 DISPATCH_LEVEL 的升高,只有在 DISPATCH_LEVEL 这个级别才会跳出线程优先级的“三界之外”。当然,有升高便会有降低,调度完了新线程要开始运行了,IRQL 自然也要降下来。PASSIVE_LEVEL 与用户态还是内核态没有必然联系,在内核态中,线程也有可能是 PASSIVE_LEVEL 优先级的。

➢ **APC_LEVEL**:用于执行异步过程调用(APC)的 IRQL。APC 的详细介绍放在后文中,这里只需要知道 APC 是在该级别运行就可以了。

➢ **DISPATCH_LEVEL**:这个级别用于线程切换和执行 DPC(延迟过程调用)。在这个级别运行,不允许发生缺页中断等事件,因为不允许发生等待事件,故在这个级别运行的线程代码都使用非分页内存以避免页面倒换的等待过程。

硬中断主要用于系统时钟、处理器间协同中断、设备中断等场景。其中设备中断对应的中断号只有几十个,因此一般一个中断号对应多个设备的中断服务例程(ISR)。

上一章在预备知识中曾经介绍过,线程切换时操作系统中有些数据结构是要切换的,如图 5-2 所示,具体包括:

➢ 任务状态段 TSS 中的 I/O 权限位图(主要是在 VM86 模式下有用,绝大多数线程的 I/O 权限位图都一样)和 ESP0 这两个域(所有线程的内核态堆栈段选择子都是一样的,因此 SS0 不需要更换)。

➢ GDT 中的 TEB 和 LDT(如果线程所属进程需要的话)要更新。

➢ CR3 寄存器要更新,CR3 寄存器中存放了当前线程内存空间地址表的物理地址,切换了 CR3 寄存器也就是切换了用户的内存空间。具体细节我们在后文中讲述。

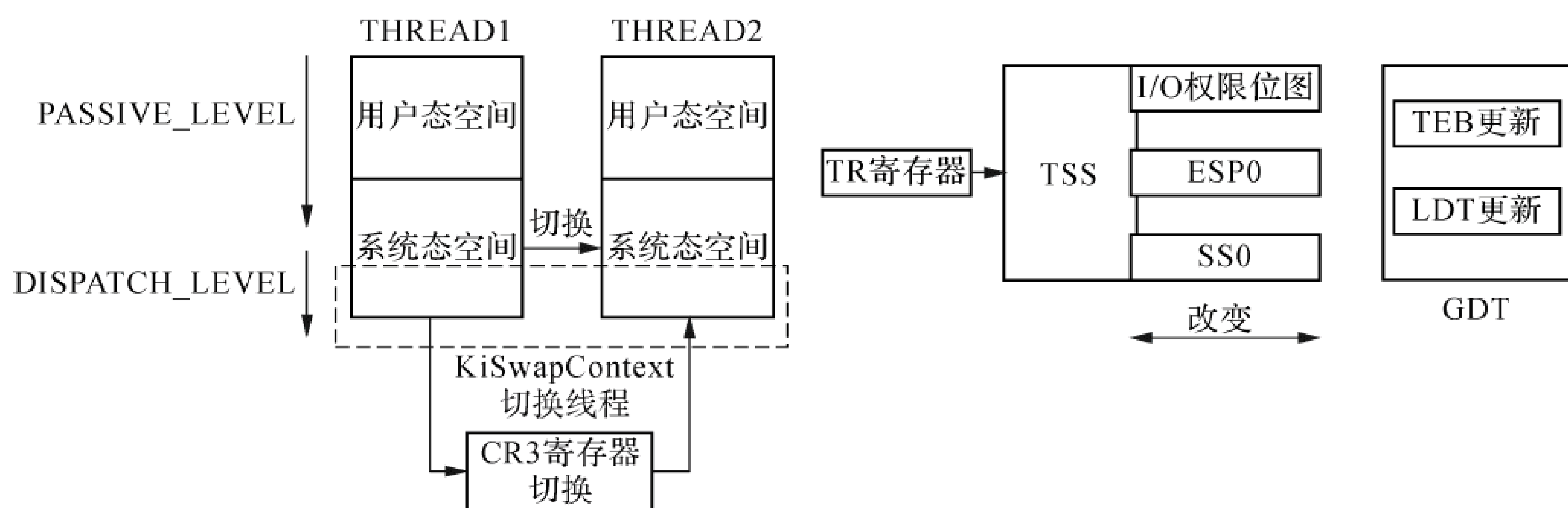


图 5-2　线程切换示意图

这里要强调的是,只要发生了 DISPATCH_LEVEL 级别或以上的中断请求,无论当时线程处于什么级别,哪怕线程的时间片没有用完,都会被抢占,因为 Windows 就是一个抢占式操作系统。中断发生时,中断服务例程(ISR)首先执行 IRQL 非常高的例程的前半段,之后会把后半段作为低 IRQL 的延迟过程调用(DPC)并使之参与到 DISPATCH_LEVEL 级别的线程调度中来。DPC 机制后文会有详细介绍。

5.2　线程切换过程

线程切换是由系统中的 KiSwapContext 函数完成的。此处我们不会讲述具体代码细节，而是用流程图、结构图的方式进行阐述，比较直观。KiSwapContext 的入参非常简单，分别是当前线程和需要切换的新线程的 KTHREAD 结构指针，这两个指针分别被放入 ECX 和 EDX 寄存器中，通过 FastCall 的方式传入 KiSwapContext。

图 5－3 是线程切换时堆栈演变的示意图，我们用 Current Thread 来表示当前线程，用 New Thread 表示要切换的新线程。

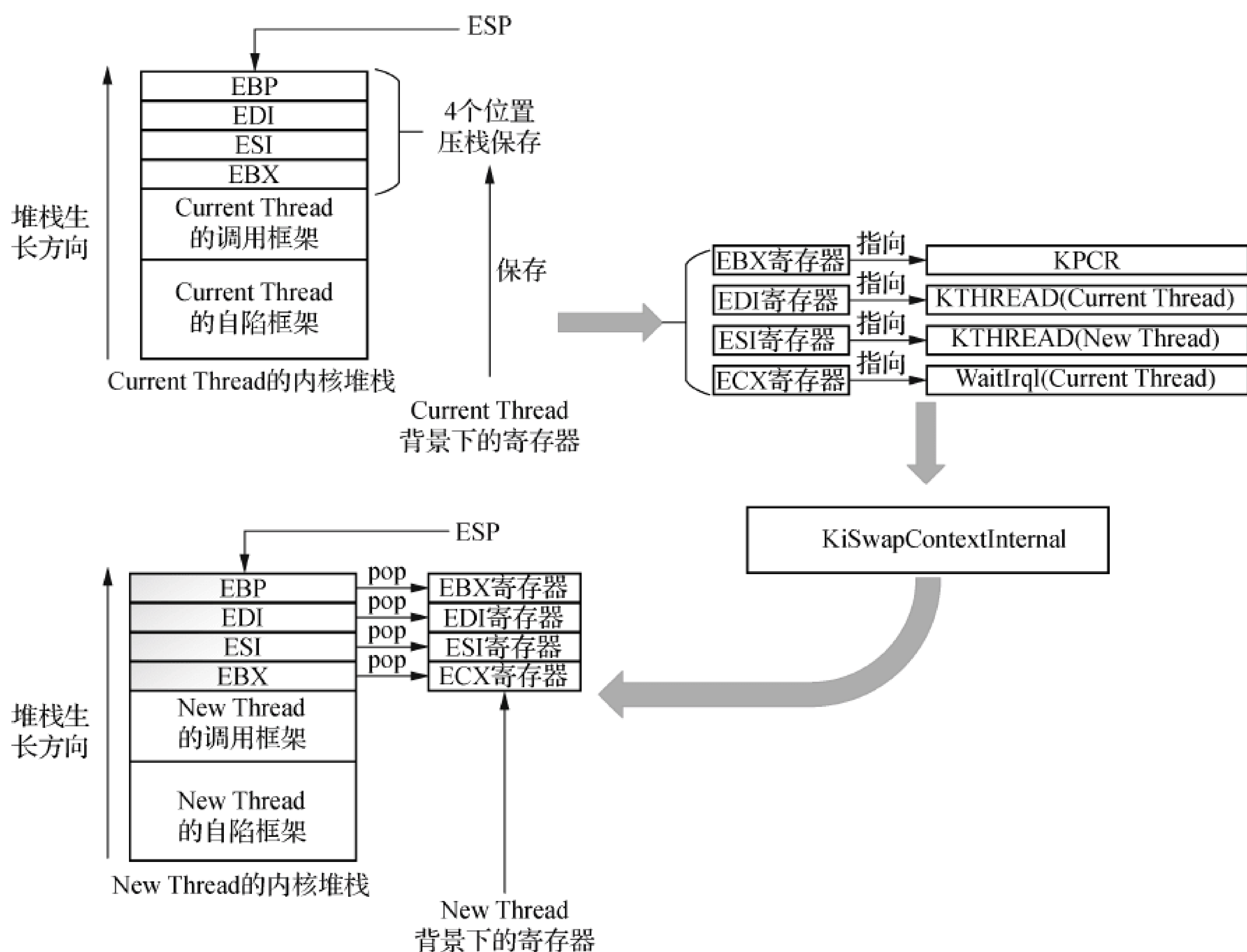


图 5－3　线程切换时堆栈的演变

在图 5－3 中我们可以看到当前线程在运行时的内核态堆栈框架，包括自陷框架和调用框架。在被切换的时候，KiSwapContext 将 4 个寄存器的值压入当前线程的系统堆栈，因为接下来要用到这 4 个寄存器，先将这些寄存器的当前值保存起来。

接下来，我们这样来安排以下寄存器：

- 使 EBX 寄存器指向当前 CPU 的 KPCR 结构；
- 使 EDI 寄存器指向当前线程的 KTHREAD 结构（从 KiSwapContext 传进来的时候是使用 ECX 寄存器的，因此 ECX 寄存器从此可以“解放了”）；
- 使 ESI 寄存器指向新线程的 KTHREAD 结构；
- 使刚刚被解放的 ECX 寄存器指向当前线程的 WaitIrql 域。

然后执行 KiSwapContextInternal，进行线程切换和堆栈切换。

执行完成后已经是新线程的天下了（内存空间、GDT 的相关域、TSS 的相关域都是新切换线程的），这时将新线程内核态堆栈栈顶的 4 个值 pop 到对应的寄存器里，如此，堆栈中只剩下该线程上次执行时留下的调用框架和自陷框架了。

由此可见，每次线程切换的时候，老线程（当前线程）堆栈中除了自己的调用框架和自陷框架，还包括了 4 个寄存器的值，这些是老线程的执行现场和若干寄存器在那个时刻的值，就像那个时刻被冰封了一样。当切换到一个新线程的时候，新线程堆栈中也是这些东西，因为它曾经也是老线程，也曾经被剥夺过执行的权利，并连同执行现场和 4 个寄存器都被压入自己的系统堆栈中。“年年岁岁花相似，岁岁年年人不同”，只有保证堆栈结构和内容安排的“花相似”，才能确保线程切换的“人不同”。

下面我们来看一下线程切换的实际操作者 KiSwapContextInternal，主要执行流程如图 5－4 所示。

图 5－4　KiSwapContextInternal 的主要执行流程

KiSwapContextInternal 执行中的主要流程如下：

（1）在当前线程（老线程）的内核态堆栈中压入本线程的 WaitIrql 和当前 KPCR 中的异常队列 EXCEPTION_LIST 的地址。其中 WaitIrql 域记录在 KTHREAD 结构中，表示原先的 IRQL。

（2）更新当前线程的内核态堆栈指针（KTHREAD. KernelStack）为当前的 ESP 寄存器的值。

（3）切换堆栈：

➢ ESP 寄存器赋值为新线程的内核态堆栈指针（KTHREAD. KernelStack）；

➢ EBP 寄存器赋值为新线程所挂靠进程的 KPROCESS 结构（进程挂靠的概念在后面会详细讲述）；

➢ EAX 寄存器赋值为当前线程所挂靠进程的 KPROCESS 结构。

（4）两个 KPROCESS 结构相比较：

➢ 如果相同，则证明新老线程是相同进程中的两个线程（内存空间一样，LDT 一样，则 CR3 寄存器与 GDT 中的 LDT 描述符均不需要切换）；

➢ 如果不同，则证明新旧线程不在相同进程中，先根据条件判断是否要更新 GDT 中的 LDT，继而更新 CR3 寄存器为新进程的 DirectoryTableBase（页目录表的物理地址），以切换内存空间。

至此，内存空间切换完成，ESP 寄存器更新完成（内核态堆栈切换完成），LDT 更新完成，新线程成为了当前线程。当前线程（新线程）的内核态栈顶结构如图 5－5 所示，这是上次线程切换时留下来的现场。

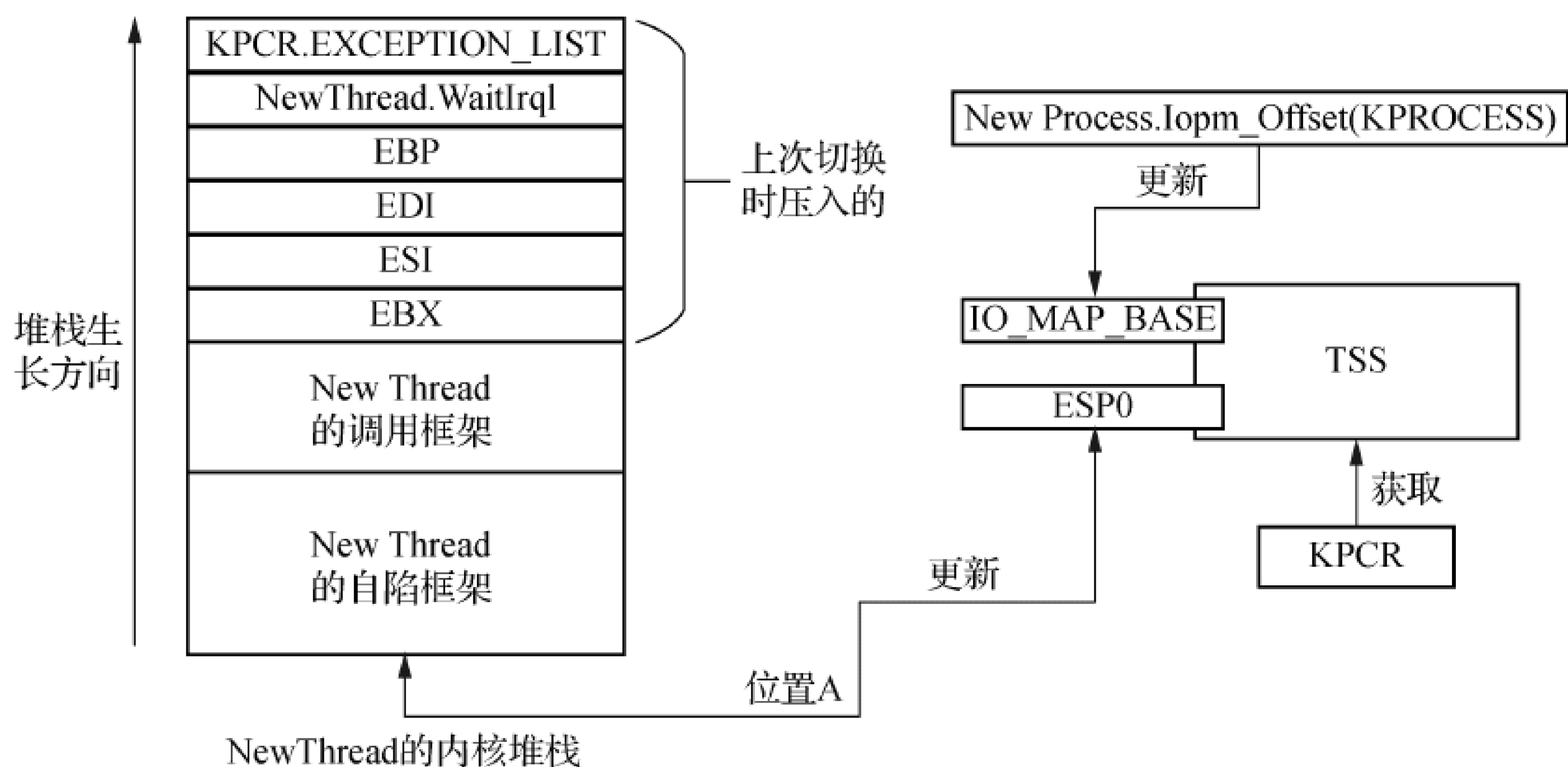


图 5－5　新线程堆栈示意图

新线程首先要执行下面两个步骤，可以看作 KiSwapContextInternal 执行的尾声：

（1）将上次切换时压入的 6 个值分别弹出到对应的数据结构中。

（2）通过 KPCR 找到 TSS（任务状态段），更新 TSS 中的 I/O 权限位图和 ESP0。

至此 KiSwapContextInternal 全部执行完毕，当前线程已切换为新线程。

综上所述，线程切换中最主要的是系统堆栈的切换，其次是某些全局数据结构中个别域

值的切换。由于有了系统调用的技术基础，线程切换过程不难理解。

本章小结

本章首先介绍了线程调度的相关概念，包括线程状态、调度优先级、中断优先级等，继而以堆栈平衡为主线详细介绍了线程的切换过程。

第 6 章　异步过程调用机制

Windows 之所以要支持异步过程调用(Asynchronous Procedure Call,APC)机制,是因为在一些场景下需要一种延迟处理的方法,特别是在内核态时有一些工作需要留在用户态处理,或者有一些不需要立即返回结果的非同步的回调处理。例如前文介绍过的线程初始化过程需要加载和连接除 ntdll. dll 外的其他动态库时就使用了 APC 机制;再比如我们在调用网络收发接口或者文件读写接口(NtReadFile)的时候,往往无法同步地返回数据报文(同步会阻塞当前线程),这些数据一般是通过“回调”的方式返回给应用软件调用方的,这种回调方式就是采用 APC 机制实现的。例如 NtReadFile 接口中的 ApcRoutine 和 ApcCONTEXT 就是与 APC 相关的参数,如图 6-1 所示。

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtReadFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,
  PULONG           Key
);
```

图 6-1　NtReadFile 接口的参数描述

本章我们将按照图 6-2 所示的提纲进行介绍。

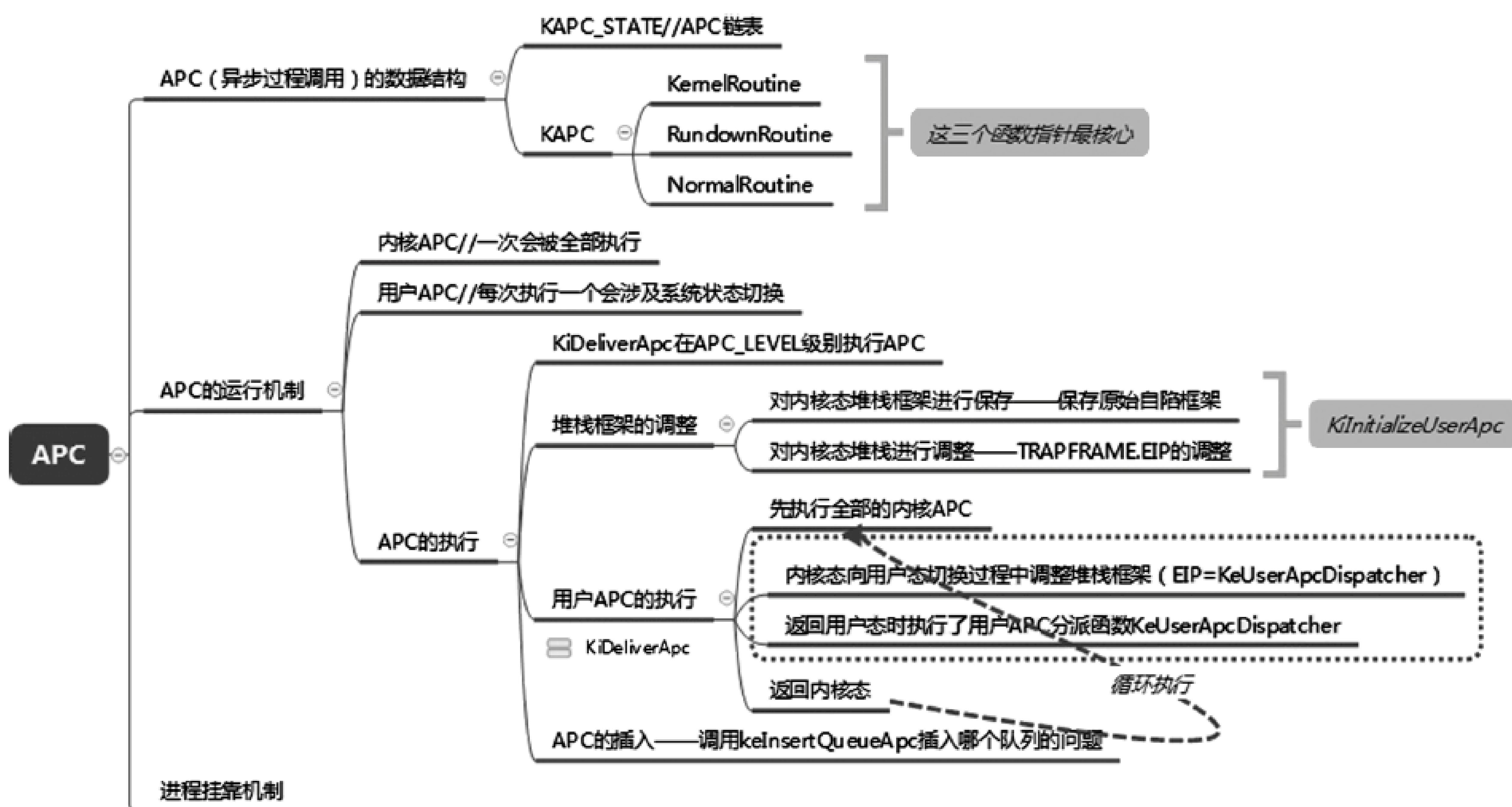


图 6-2　本章提纲

6.1 APC 的数据结构

APC 是与线程相关的,发起 APC 请求的只能是当前线程,但是目标线程则不一定是当前线程,有可能是子进程中的线程(例如 LdrInitilizeThunk 的 APC 请求),也有可能是与自己不相关的其他线程(远程线程)。Win32 API 中的 QueueUserAPC 方法是用户态线程发起 APC 请求的接口,它的名称也很好地诠释了 APC 在线程中的存在形态:队列中存储。在 KTHREAD 结构中也存在与 APC 相关的队列和索引等信息。每个线程都有若干个 APC 队列,之所以有多个队列,一是因为 APC 有内核 APC 和用户 APC 之分,二是进程挂靠时要将原生的 APC 队列挪到他处保存起来,腾出位置来保存挂靠进程环境下的 APC 请求,这都需要队列的支持,KTHREAD 结构中与 APC 相关的字段如下所示:

```
typedef struct_KTHREAD
{
    KAPC_STATE ApcState;                //该数据域包含了当前的两个 APC 队列,以及所属进程和 APC 执
                                        //行状态
    SHORT KernelApcDisable;             //是否禁用内核 APC
    UCHAR ApcStateIndex;                //表示一个枚举值,OriginalApcEnvironment 表示当前 APC
                                        //是原生态;AttachedApcEnvironment 表示当前 APC 是挂
                                        //靠态
    PKAPC_STATE ApcStatePointer[2];     //ApcState 的指针,在原生 APC 状态下,0 号元素指向 ApcState,
                                        //1 号元素指向 SavedApcState;在挂靠 APC 状态下则反过来
    KAPC_STATE SavedApcState;           //与 ApcState 结构相同,存储了挂靠环境下的原生 APC
} KTHREAD, * PKTHREAD;
```

我们再来看看 ApcStateIndex 的枚举值和 KAPC_STATE 结构,以方便我们理解 APC 队列,一般情况下 ApcStateIndex 会使用 OriginalApcEnvironment 和 AttachedApcEnvironment 两个值,如下所示:

```
typedef enum_KAPC_ENVIRONMENT {
    OriginalApcEnvironment,      //原始的进程环境
    AttachedApcEnvironment,      //挂靠后的进程环境
    CurrentApcEnvironment,       //当前环境
    InsertApcEnvironment         //被插入时的环境
} KAPC_ENVIRONMENT;

typedefstruct_KAPC_STATE {
    LIST_ENTRY ApcListHead[MaximumMode];   //当前线程的内核和用户两种状态的 APC 链表
    struct_KPROCESS * Process;             //当前线程所属进程的 EPROCESS 指针,
                                           //PsGetCurrentProcess()就返回此值
    BOOLEAN KernelApcInProgress;           //表示内核 APC 正在执行
    BOOLEAN KernelApcPending;              //表示内核 APC 正在等待执行
    BOOLEAN UserApcPending;                //表示用户 APC 正在等待执行
} KAPC_STATE, * PKAPC_STATE, * PRKAPC_STATE;
```

上述数据结构也无须过多解释。APC 数据结构则如下所示:

```
typedef struct_KAPC {
CSHORT Type;
    CSHORT Size;
    ULONG Spare0;
    struct_KTHREAD * Thread;
    LIST_ENTRY ApcListEntry;              //用于挂入 KAPC_STATE 中的链表
    PKKERNEL_ROUTINE KernelRoutine;       //内核模式 APC 执行函数的指针
```

```
    PKRUNDOWN_ROUTINE RundownRoutine;       //线程终止时还有 APC 没执行则会调用这个函数
    PKNORMAL_ROUTINE NormalRoutine;         //指向用户提供的 APC 执行函数。0 表示是一个特殊内核
                                            //APC,否则是一个普通的(内核态/用户态)APC。特殊
                                            //APC 位于链表前部,普通的位于链表尾部
    PVOID NormalContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    CCHAR ApcStateIndex;                    //APC 环境状态,OriginalApcEnvironment 或
                                            //AttachedApcEnvironment
    KPROCESSOR_MODE ApcMode;                //表示该 APC 是内核态还是用户态
    BOOLEAN Inserted;
} KAPC, * PKAPC, * RESTRICTED_POINTER PRKAPC;
```

从 APC 结构可以看出,KernelRoutine、RundownRoutine 和 NormalRoutine 都是函数指针:

- **RundownRoutine**:用于扫尾,也就是针对线程结束时 APC 队列不为空的情况。
- **KernelRoutine**:表示内核态 APC 函数,一般由内核线程发起。
- **NormalRoutine**:指向用户态 APC 函数的总入口,即 kernel32. dll 的内部函数 IntCallUserApc,与线程初始化类似,这是普通 APC 函数的总入口,而接下来的 NormalContext 才真正指向用户线程设置的函数入口,远观过去,就好像是 IntCallUserApc 把用户指定的函数又封装了一层,这也是为了将用户指定的函数纳入结构化异常保护的框架之下。

因此我们可以这样理解:APC 本质上是一个回调函数,这个回调函数不需要立即返回,而是等着有结果或时机成熟以后再返回(这也是回调函数的要义)。但与一般的回调函数不同,APC 不局限于为本线程设置回调,也可以为包括子进程线程和远程线程在内的其他线程设置回调。回调函数的执行时机也与一般回调不同,除了要满足相关条件还要有一定的时机选择:

- 用户态线程在从内核态返回用户态的过程中;
- 内核态线程在中断请求级别降低或线程切换的时候。

6.2　APC 的运行机制

6.2.1　APC 的执行流程

下面我们以 APC 的执行流程为主线详细讲述 APC 机制的原理。前文讲过,在系统调用的尾声要执行 KiServiceExit 返回用户态空间。在 KiServiceExit 的执行过程中有个宏定义 CHECK_FOR_APC_DELIVER,从字面上来解释也比较开宗明义:对投递的 APC 进行检查和执行。其对应的系统函数为 KiDeliverApc,我们就从这里展开。

KiDeliverApc 函数要求在 APC_LEVEL 中断请求级别上执行,其执行流程如下:

1）步骤 1:执行内核模式 APC 队列中的所有 APC

针对每个 APC,执行 KernelRoutine 函数,并且以 NormalRoutine 为参数,执行完成后检查 NormalRoutine 是否为空(执行完 KernelRoutine 后,NormalRoutine 也可能为空了):

- 若不为空，需要先将 IRQL 降到 PASSIVE_LEVEL，再执行 NormalRoutine，执行完还需要再回到 APC_LEVEL 级别，然后执行步骤 2；
- 若为空则直接执行步骤 2。

2）步骤 2：执行用户模式 APC 队列中的首个 APC

步骤 2 以 NormalRoutine 为参数执行 KernelRoutine。

- 若 KernelRoutine 执行完毕 NormalRoutine 为空，则尝试唤醒正在等待的线程，且必须是可唤醒的线程。
- 若 KernelRoutine 执行完毕 NormalRoutine 不为空，则需要为在用户态空间执行 APC 做一番准备：执行 KiInitializeUserApc，在用户态堆栈中安排一番，并于预返回用户态空间后的指令入口，使我们投递的用户模式 APC 得到执行。

KiDeliverApc 的执行流程如图 6－3 所示。

图 6－3　KiDeliverApc 的执行流程

注:假如线程在某时刻调用了 Sleep,那么在超时时间点到来之前,无论怎么样该线程都不会被唤醒以继续执行;假如将 Sleep 换作 SleepEx,哪怕超时时间点尚未到来,我们仍然可用 KeTestAlertThread 方法将其唤醒以使其继续执行。我们将前者称为不可唤醒,将后者称为可唤醒。

在执行完 KiInitializeUserApc 或者 KeTestAlertThread 后,KiDeliverApc 函数就执行结束了。此处留下了一个伏笔,就是 KiInitializeUserApc 在线程的内核态堆栈中构筑了一个伪框架,用于干涉回到用户态空间后的执行步骤。下面我们来看看 KiInitializeUserApc 具体是怎样安排系统堆栈框架的。

6.2.2　对堆栈框架的安排

1）ESP 指针在系统堆栈中的指向

如图 6－4 所示,执行 KiServiceExit 时 ESP 寄存器是指向 TRAP_FRAME 的起始地址的,这是因为 KiServiceExit 是在内核态的最后一步,但凡执行到这一步,内核态堆栈中 TRAP_FRAME 以上的框架都已经消耗掉了。而从 KiServiceExit 到 KiInitializeUserApc,内核态堆栈没有变化,这就意味着执行 KiInitializeUserApc 的时候 ESP 寄存器还是指向 TRAP_FRAME,也就是指向了内核态堆栈中保存的线程最近一次进入内核前完整的用户态空间的寄存器状态。这些状态包括返回用户态空间后的指令指针、堆栈指针等信息。因此我们可以操纵这些指针,使其指向任何我们想要的地址。

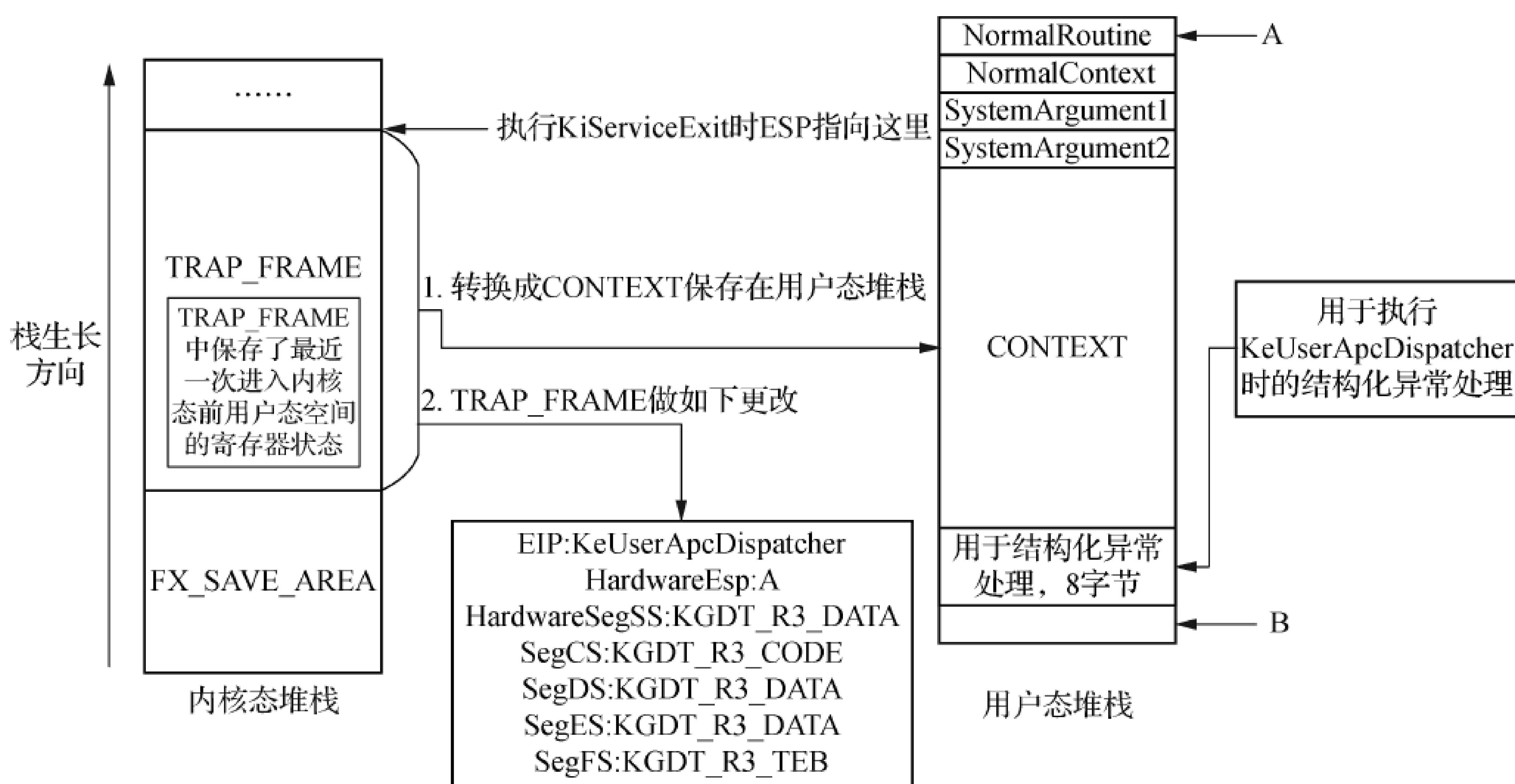


图 6－4　KiInitializeUserApc 对于线程堆栈的操作

2）对内核态堆栈 TRAP_FRAME 的保存

如图 6－4 所示,KiInitializeUserApc 首先将内核态堆栈中的 TRAP_FRAME 保存到用户态堆栈中。TRAP_FRAME 本来就保存了上次切换的寄存器信息,那为何还要再保存到用户

态堆栈呢？这是因为我们要对内核堆栈中的 TRAP_FRAME 做更改，因此只能将原来的 TRAP_FRAME 保存在用户态堆栈中。这里做了一下转化，用户态堆栈是这样调整的：

- 将 TRAP_FRAME 转化为 CONTEXT 存放在用户堆栈中。
- CONTEXT 的上面（堆栈低址）初始化了 4 个位置，用于存放 KeUserApcDispatcher 的参数：NormalRoutine、NormalContext、SystemArgument1 和 SystemArgument2。
- CONTEXT 的下面（堆栈高址）要空出 2 个位置用于 KeUserApcDispatcher 的结构化异常处理框架。这样一来，图 6－4 中的位置 A 就成了用户态堆栈的新指针，而 B 则是老的堆栈指针。

3）对内核态堆栈 TRAP_FRAME 的更改

接下来对 TRAP_FRAME 进行更改，EIP 指令指针修改为 ntdll. dll 中的 **KeUserApcDispatcher**，这是 APC 投递函数的总入口，主要步骤包括：

（1）在用户态堆栈中构筑 KeUserApcDispatcher 执行时的结构化异常处理框架，其中异常处理函数为 KiUserApcExceptionHandler，这是专门用于 APC 执行保护的函数。

（2）以 NormalContext、SystemArgument1、SystemArgument2 为参数调用 NormalRoutine，就是前文提到过的 kernel32. dll 中的 IntCallUserApc，而 NormalContext 中才是真正的用户指定 APC 回调过程地址，SystemArgument1 和 SystemArgument2 都是 NormalContext 的参数。

（3）调用 ZwContinue 返回内核态，继而循环执行整个用户 APC 队列。

完成了堆栈的更改后 KiInitializeUserApc 的使命也完成了，“万事俱备只欠执行”，可以预见，当线程通过 KiServiceExit 返回用户态空间后，执行的第一条指令就是 KeUserApcDispatcher（TRAP_FRAME. EIP 的值），其作用是在用户态空间中执行全部的用户 APC。

前文说过，KiInitializeUserApc 执行到尾声时要调用 ZwContinue 返回内核态空间，要保证就像没有执行过 APC 一样还原系统堆栈的框架。幸好有先见之明，KiInitializeUserApc 将原生的 TRAP_FRAME 保存在了用户态堆栈，现在是该还原的时候了。

ZwContinue 将用户态空间堆栈的 CONTEXT 转换回 TRAP_FRAME 并压入内核态堆栈，再加上 ZwContinue 的调用框架，就形成了图 6－5 中内核态堆栈的格局。从另一个角度来看，ZwContinue 就好像是一次新的系统调用，使当前线程再次“陷入”系统空间中。

同时，由于 ZwContinue 将上次的自陷框架还原，因此 APC 执行之前的用户态堆栈指针 B 也被还原了回去（在内核态的 TRAP_FRAME 中），从而使用户态堆栈被配平，恢复了“往日的平静”，如图 6－5 所示。

当然，ZwContinue 作为系统调用，总有再次从内核态回到用户态的时候，而回去的契机就是执行 KiServiceExit，执行这一步，也就意味着 IRQL 会再次进入 APC_LEVEL，从而继续执行 APC。这就比较有意思了，过往的流程像一个环路又回到了记忆中的原点。

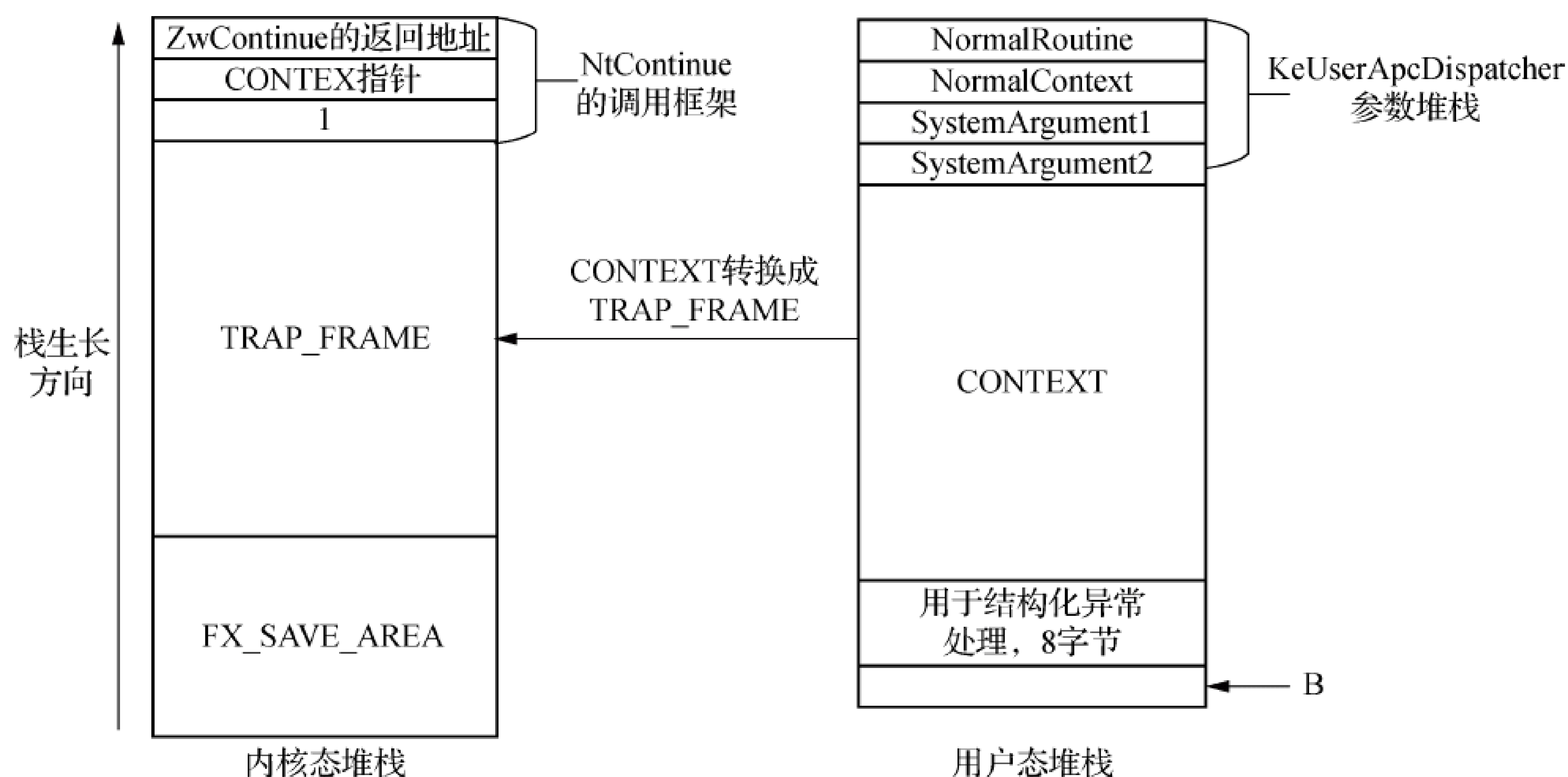


图 6-5　调用 ZwContinue 时的线程堆栈

6.2.3　用户 APC 的执行流程

图 6-6 是用户 APC 的执行流程，可以看出其执行脉络如下：

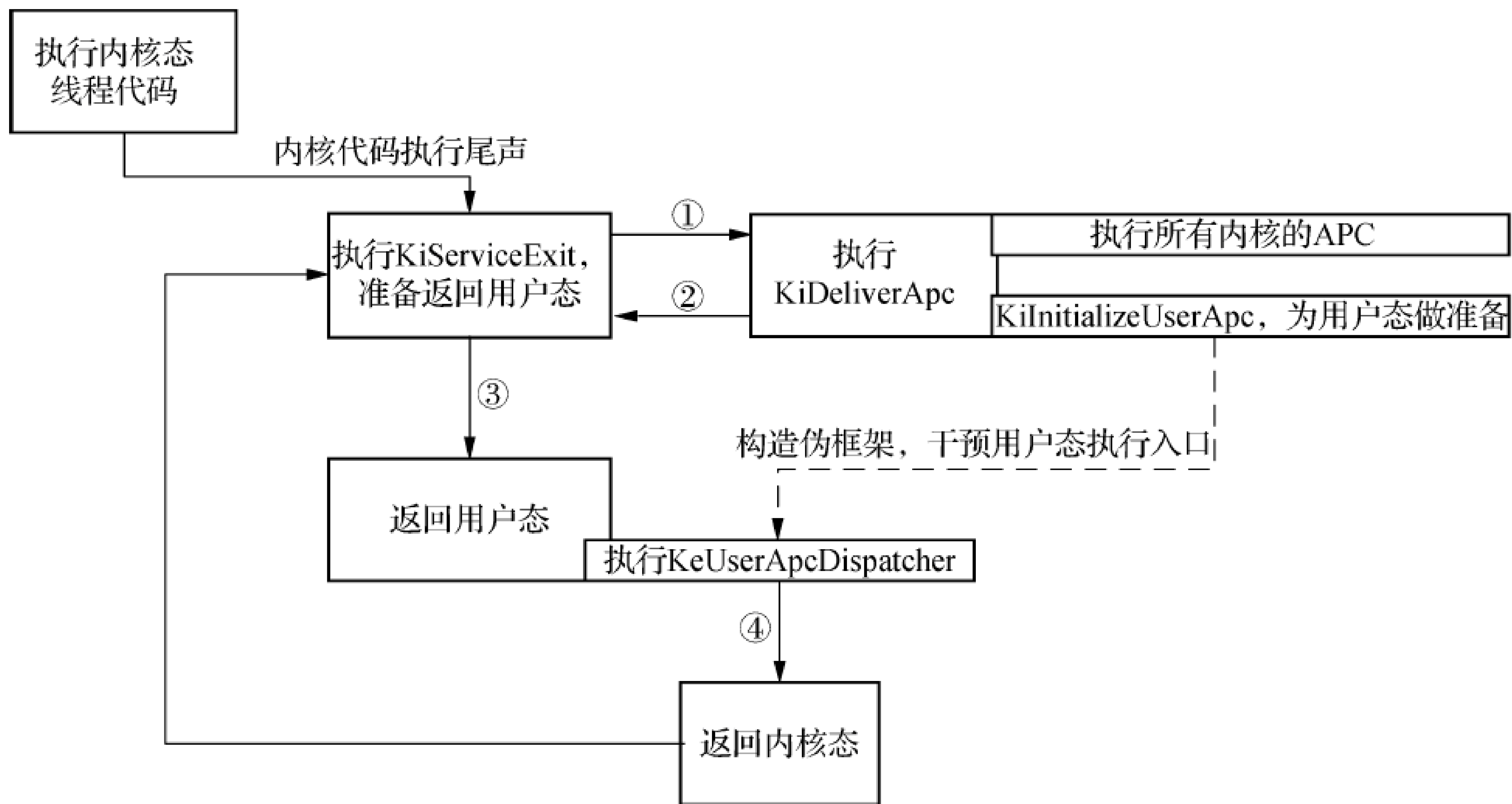


图 6-6　用户 APC 的执行流程

（1）内核态代码在即将返回用户态时（在尾声执行返回用户态的函数 KiServiceExit）会执行 KiDeliverApc（①）。

（2）KiDeliverApc 首先执行线程中所有的内核 APC，并且调用 KiInitializeUserApc 构造内核堆栈中的伪框架，为用户态执行做准备。

（3）当从 KiDeliverApc 返回时（②），也是切换到用户态的关键时刻（③）。

(4) 在用户态中按照事先被干预的自陷框架(EIP)执行 KeUserApcDispatcher,这是用户 APC 的总入口。

(5) 第一个用户 APC 执行的尾声要调用 ZwContinue 来再次返回内核态(④),作为系统调用来讲,ZwContinue 又重新构造了内核态堆栈的布局。

(6) 当内核态将要切换到用户态时再次调用 KiServiceExit,循环往复又回到了原点,一轮用户 APC 流程处理完毕。

再次执行的时候 KiDeliverApc 不再需要执行内核 APC 了,因为第一轮已经把内核 APC 全部执行完了,按照上述流程能跳过的跳过,能简化的简化,再次循环,直到用户 APC 消耗完毕。可以看出,内核 APC 的执行是不需要切换堆栈的,而用户 APC 的执行则要反复切换堆栈。

6.2.4 APC 的插入

APC 的插入是采用 Win32 API QueueUserApc 来实现的,它又调用 NtQueueApcThread,并以用户 APC 总入口 IntCallUserApc 和用户软件指定的回调接口为参数。NtQueueApcThread 首先构造 APC 结构,继而调用 KeInsertQueueApc 将 APC 插入到 KAPC_STATE 的相应队列中。

那么到底插入到哪个 KAPC_STATE 的队列呢? 这是根据 APC 请求中的 ApcStateIndex 来决定的:

- 若是 OriginalApcEnvironment,则插入到 ApcState 的队列中;
- 若是 AttachedApcEnvironment,则插入到 SavedApcState 的队列中;
- 同时也要看 APC 的 ApcMode,是 UserMode 则插入到 KAPC_STATE 的用户 APC 队列;否则,插入到 KAPC_STATE 的内核 APC 队列。

这里还要区分一下 APC 的性质:如果 APC 的 NormalRoutine 为空,那么这个 APC 就是一个特殊 APC,会在对应的 KAPC_STATE(ApcState/SavedApcState)的用户模式/内核模式列表(UserMode/KernelMode)中选择第一个 NormalRoutine 为空的 APC,插入到该 APC 的前面;如果 APC 的 NormalRoutine 非空,那就是个常规 APC,会根据其他判断条件选择插入到队头还是队尾。

插入完成后也会判断一下 APC 所属线程是否为当前线程。若为当前线程,则将当前处理器的 IRQL 提升到 APC_LEVEL,相当于发出了一个 APC 的中断请求;若不是当前线程,则唤醒目标线程使之准备执行 APC。执行完这一切后,会调用 KiExitDispatcher 退出线程切换态(DISPATCH_LEVEL),并降低 IRQL,从而触发 APC 的投递(KiDeliverApc)。

前文讲过,类似 NtReadFile 这样的接口也是借用 APC 机制实现的。我们首先来看 Win32 API ReadFileEx,其最后一个参数 lpCompletionRoutine 就是需要用户指定的文件读完成后的回调接口,用于通知用户应用软件,如图 6-7 所示。

```
BOOL ReadFileEx(
  HANDLE                            hFile,
  LPVOID                            lpBuffer,
  DWORD                             nNumberOfBytesToRead,
  LPOVERLAPPED                      lpOverlapped,
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

图6-7　ReadFileEx 接口参数

ReadFileEx 调用 NtReadFile，NtReadFile 有两个参数 ApcRoutine 和 ApcContext，分别对应了内置的 APC 例程 IntCallUserApc 和 ReadFileEx 传递下来的 lpCompletionRoutine。前文讲过，lpCompletionRoutine 会被置于 IntCallUserApc 框架下调用。类似 ReadFile/WriteFile 这样的I/O 操作会被 I/O 管理器统一翻译成 IRP（I/O 请求包，在介绍驱动程序的章节中会有详细介绍）。IRP 中有两个域来传承 NtReadFile 传递下来的这两个参数：UserApcRoutine 和 UserApcContext，之后便调用 IRP 的执行函数 IoCallDriver。其实，即使执行到这里，本质上也只是将用户赋予的回调指针传了下来。

当 IRP 完成后（读完成或写完成），I/O 管理器的系统函数 IopCompleteRequest 调用 KeInitializeApc 将上面传下来的两个参数封装成 APC，并且调用 KeInsertQueueApc 将 APC 插入到线程的相应队列中（用户 APC 队列/内核 APC 队列），这就与我们前面讲的内容对接上了。调用 KiExitDispatcher 后触发了 APC 的投递，这样当从内核态向用户态切换时，APC 被执行，应用软件被回调通知。

APC 也是进程注入的常用机制，即通过 APC 向其他线程投递代码模块，线程切换 APC 代码被执行，从而劫持被投递线程的执行权，后文会详细介绍这种机制。

6.3　进程挂靠机制

最后，我们谈一谈进程挂靠机制。

所谓进程挂靠，本质上就是切换进程的地址空间，更具体地说，是切换到目标进程的用户态空间（各进程的内核地址空间基本一致），从而使当前线程的运行地址空间环境发生切换。在创建进程和为子进程创建线程的过程中，我们会用到进程挂靠机制。

进程挂靠的系统调用为 KeAttachProcess，而解除挂靠的系统调用为 KeDetachProcess。KeAttachProcess 有如下三个作用：

- 保证当前线程的系统堆栈在新进程的地址空间中有映射。
- 提高当前的 IRQL，使线程无法调度切换。
- 调用 KiAttachProcess 实施实质的挂靠动作。

而 KiAttachProcess 要完成以下几项工作：

- 改变 APC 环境，将原生的 ApcState 暂时挂入 SavedApcState 中，这样在进程挂靠的新环境下投递的 APC 都会挂入 ApcState 中，待进程挂靠回去的时候再换回来。

➢ 降低当前的 IRQL,使线程处于可以切换的中断请求级别,并回到原来的中断请求级别。
➢ 执行 KiSwapProcess,切换 CR3 寄存器,使 CR3 寄存器指向目标进程(新进程)的页面目录表物理地址(进程地址空间切换,只是切换页面目录表,不切换堆栈)。
➢ 执行 KiExitDispatcher 退出线程调度切换状态。

每当一个线程改变了优先级,或者挂入了一个 APC 请求,或者发生了挂起/恢复了一个线程的运行,或者从睡眠中唤醒了一个线程,或者退出了某个线程的切换禁区的时候,都会执行 KiExitDispatcher,其执行流程如图 6-8 所示。KiExitDispatcher 的作用是:

➢ 如果 DPC(延迟过程调用)存在,则执行 DPC。
➢ 如果 PRCB 中存在当前线程的剥夺线程(NextThread 不为空),则进行线程切换。

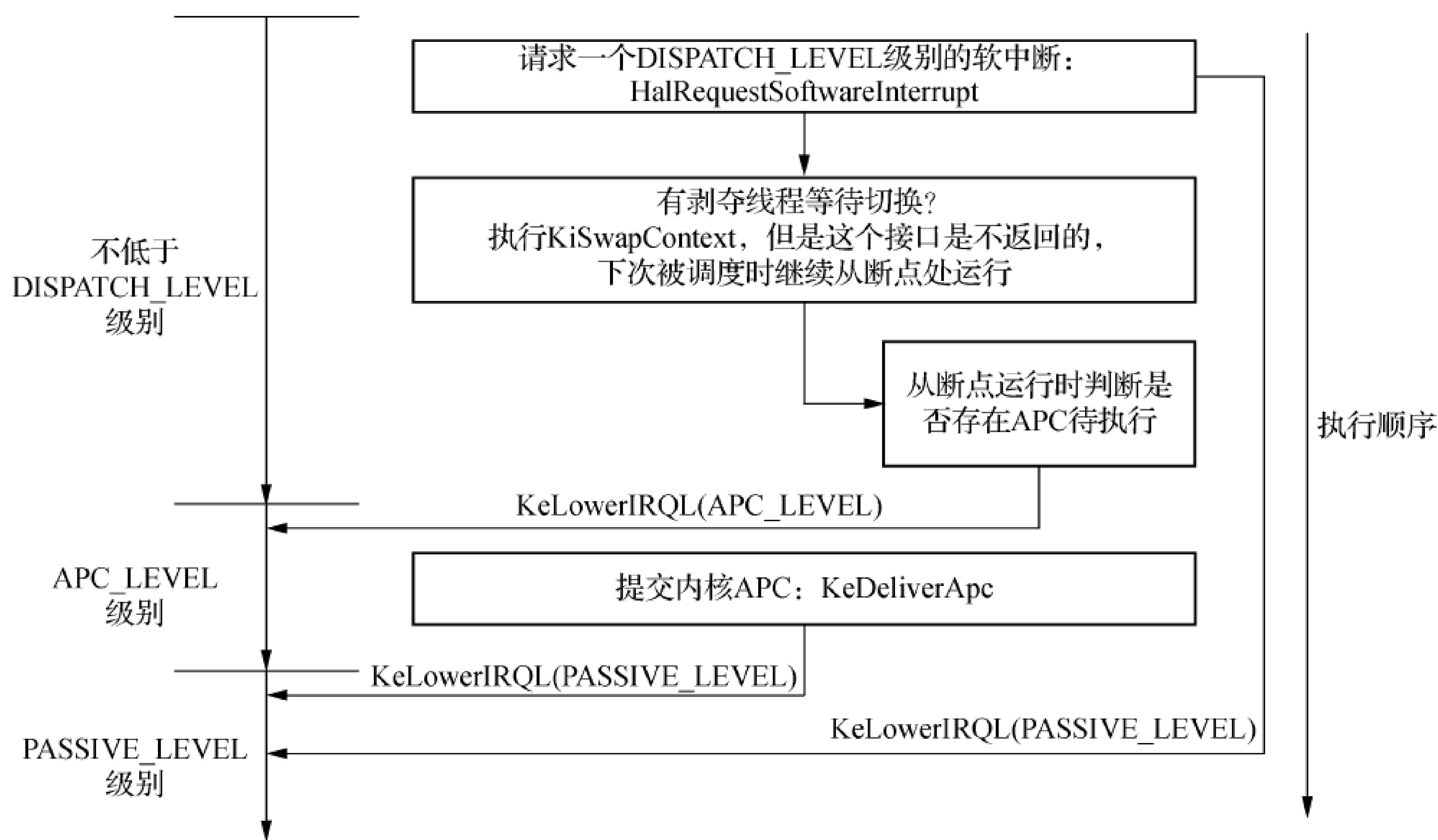


图 6-8 KiExitDispatcher 执行流程示意图

本章小结

异步过程调用(APC)是 Windows 系统中非常有用的一种机制,正如其命名所描述的一样,APC 是一种异步通知/异步结果返回的框架机制。

本章首先介绍了 APC 相关的数据结构,然后以堆栈平衡为主线讲述了 APC 的运行机制,包括总体的执行流程、在用户态和内核态对各自堆栈做出的更改和布局、返回用户态空间后的执行过程以及 APC 的插入和调度流程,最后讲述了 APC 的应用实例——Windows 进程挂靠机制。

第7章　系统中断机制

我们在使用 Windows 的时候经常会操作鼠标或键盘等 I/O 设备进行输入，或依赖网卡进行数据收发。那么在我们点击了鼠标或按下键盘按键后，或网卡接收到数据包继而希望通知应用软件时，这些触发的信号怎样被系统接收和理解呢？这就要用到系统中断机制。其实介绍系统中断机制要先从 CPU 开始说起，因为中断首先是个硬件行为（当然也有软中断，这里主要是指设备中断），在操作系统软件的作用下将这些硬件行为翻译成软件信号，并根据这些软件信号执行要求的不同来择机执行，同时还要考虑不同优先级的中断发生时的协调问题。

本章将按照图 7－1 所示的提纲来讲解中断和中断的后处理过程。

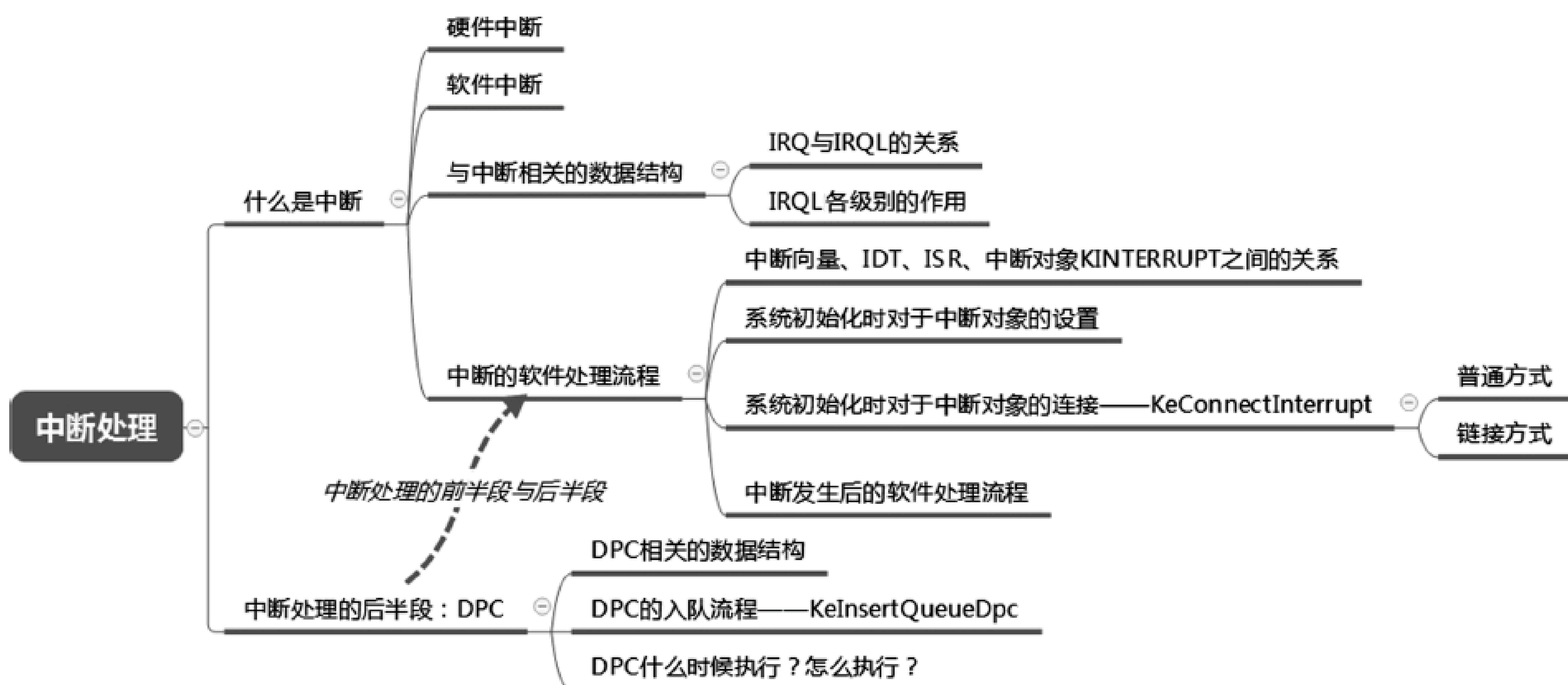


图 7－1　本章提纲

7.1　中断机制概述

7.1.1　中断的硬件处理机制

中断分为可屏蔽中断和不可屏蔽中断。在 CPU 的引脚上，INTR 中断引脚代表可屏蔽中断，NMI 代表不可屏蔽中断，这两个引脚负责中断信号的输入与输出。但是那么多的 I/O 设备，那么多的中断信号，只靠一两个引脚管理与输出也太勉为其难了。所以每个 CPU 还接有一个可编程中断控制器（PIC）或高级可编程中断控制器（APIC）来赋能，其中 APIC 更适合多处理器的情况。PIC/APIC 的作用如下：

➢ 扩展了中断信号的接入源，也就相当于扩展了 I/O 中断设备的接入量；

➢ 将 I/O 设备输入的中断信号根据优先级向 CPU 发起串行的中断请求，并且屏蔽低优先级的信号，有点类似以消息队列的方式仲裁中断并发。

例如耳熟能详的 8259A 芯片就是典型的 PIC，多片 8259A 可以级联，构成主从模式，主片的 INT 引脚连接 CPU，从片的 IR_0—IR_7 引脚连接到 I/O 设备，而主片与从片之间的连接链路是从片的 INT→主片 IR_0—IR_7，从而扩展了外设的接入量。

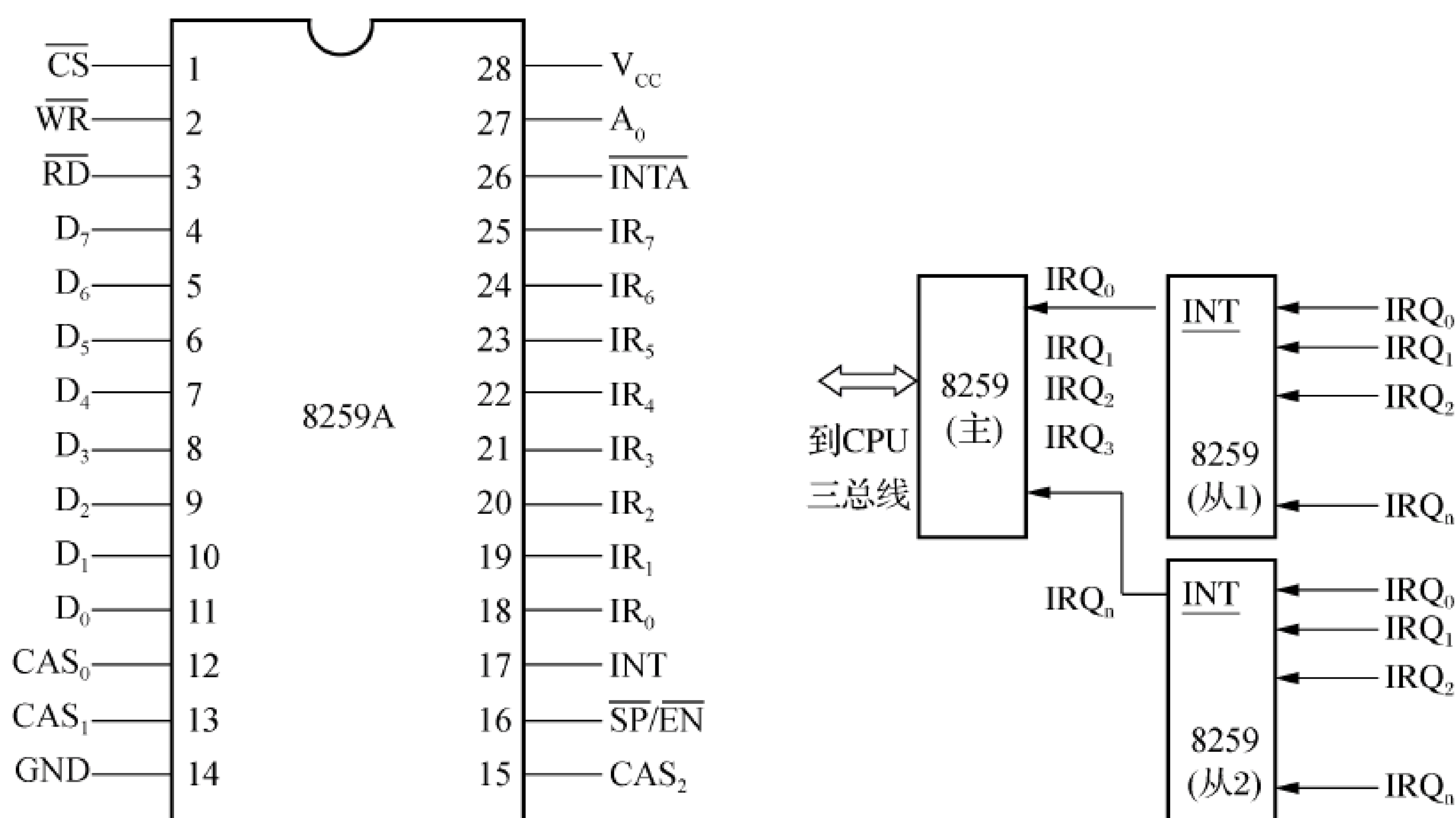


图 7－2　8259 芯片的构造与级联示意图

图 7－2 中 8259A 芯片的中断优先级别默认是根据中断输入引脚 IR_0→IR_7 逐个递减的，也就是说连接到 IR_0 引脚上的设备的中断优先级别最高，IR_7 引脚上的最低，IRQ 根据优先级分别输入到对应的 IR 引脚。当然，编程控制器的引脚中断优先级别完全可以通过“编程”方式来定义，这里不再赘述。

APIC 由如下两部分组成：

➢ **本地 APIC（LAPIC）**：与 CPU 绑定，用于控制传送给逻辑处理器的中断信号，并产生处理器间中断（IPI），接收本地中断源（例如从 I/O APIC 传过来的信号、处理器间中断信号、APIC 定时器中断信号等）。

➢ **I/O APIC**：与外部设备控制器绑定，用于接收外部设备的中断。

图 7－3 展示了 LAPIC 与 I/O APIC 的关系和连接方式。

中断是个由 I/O 设备、中断控制器、CPU 和操作系统软件共同作用的机制。在 PIC/APIC 中，中断请求（IRQ）与 Windows 中的 IDT 是一一对应的，因此中断的本质就是 PIC/APIC 的 IRQ 和与之对应的中断处理例程之间的协作。

但是我们在前面介绍过 IRQL（中断请求级别），那么 IRQ 与 IRQL 是什么关系？为什么“长”得那么像？原来，Windows 为自己量身打造了一套中断请求优先级方案 IRQL，这是一套纯软件的中断请求优先级方案，与 IRQ 的关系是线性对应的，只是 IRQL 也囊括了软件中断和最低的软件运行请求级别（PASSIVE_LEVEL），比 8259A 芯片的 IRQ 机制更完善。那么从更宏观的角度来说，IRQL 机制也更好地保证了 CPU 不被比当前请求级别低的中断事件打扰。

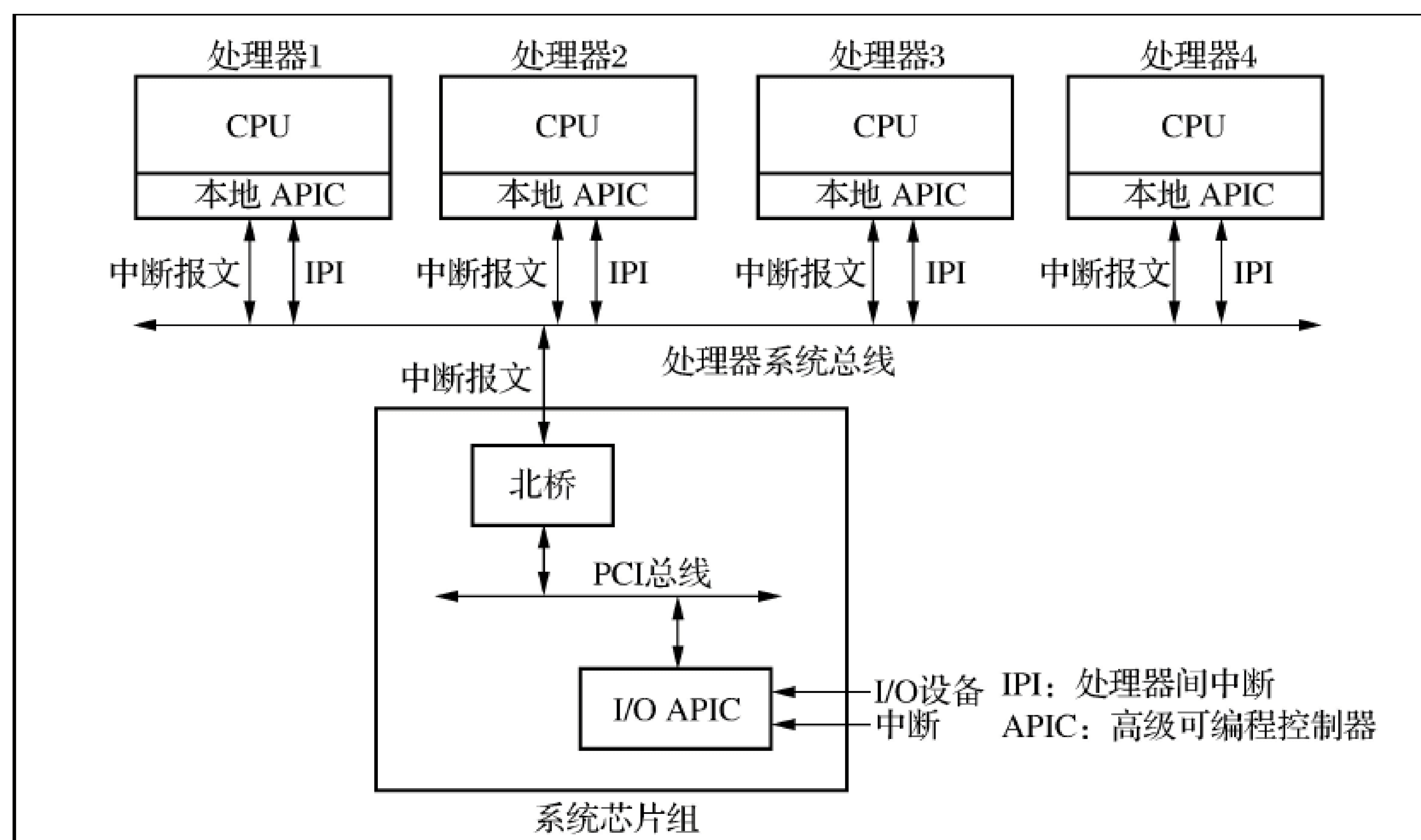


图 7－3　LAPIC 与 I/O APIC 的关系和连接方式

中断的处理过程(硬件处理 + 软件处理)如图 7－4 所示，由于图示已经展示得比较清楚，在此不再详细叙述。

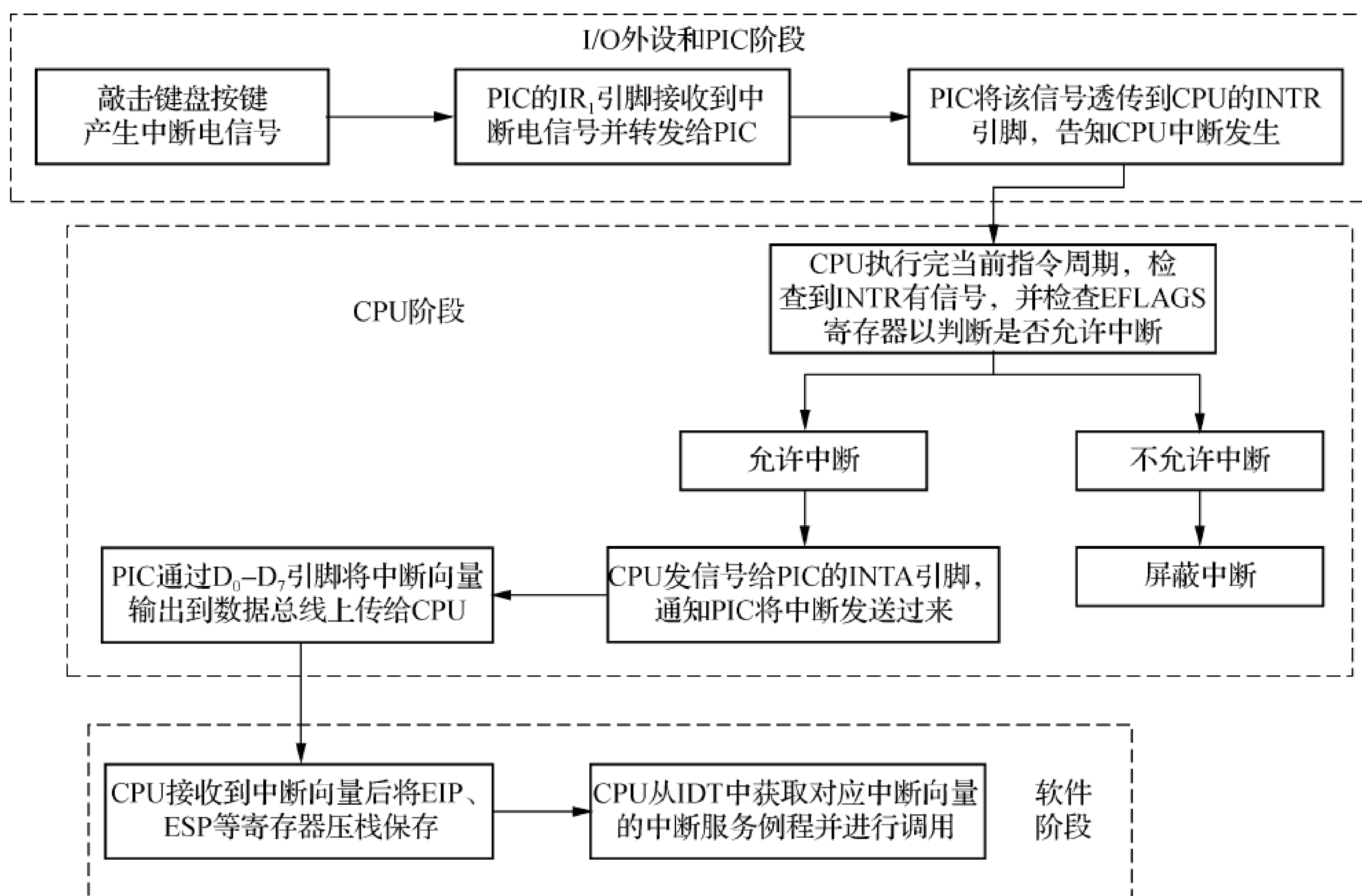


图 7－4　中断处理过程

接下来介绍中断的软件处理环节。

7.1.2 中断的软件处理机制

要处理中断,必须有操作系统软件的配合。其实 PIC/APIC 中输出的这些中断信号的处理过程是相似的,但是每种信号对应的软件处理过程肯定是不一样的,否则只定义一种信号就行了,因此这些从引脚输入的中断信号经过 CPU 翻译后就变成了“中断向量”,其代表了某信号在 IDT(中断描述符表)中对应的 ISR(Interrupt Service Routine,中断服务例程)的下标。当然,这些硬件中断信号(或者叫中断向量)只是在 IDT 中映射了一部分表项,而包括一些软件中断(例如自陷 int 0x2E、断点 int 0x03)等也要在 IDT 中占用一部分中断向量。Windows 在启动的时候对 PIC/APIC 进行设置,包括 PIC/APIC 芯片的工作模式、IRQ 与 IDT 的映射关系等都是在这一阶段完成的。

IRQ 与 IDT 的映射过程在 Windows 中被称为中断服务例程的“连接”,这是系统初始化的重要步骤。在 Windows 中有个表示中断对象的数据结构 KINTERRUPT,一个 KINTERRUPT 对应着一个 CPU。KINTERRUPT 就是为中断连接服务的,其具体域如下所示:

```
typedef struct _KINTERRUPT {
    CSHORT Type;
    CSHORT Size;
    LIST_ENTRY InterruptListEntry;        //用于挂入不同中断向量的中断列表
    PKSERVICE_ROUTINE ServiceRoutine;     //中断服务例程,一般指向 KiInterruptDispatch
    PVOID ServiceContext;                 //中断服务例程的参数
    KSPIN_LOCK SpinLock;                  //用于同步的自旋锁
    ULONG TickCount;
    PKSPIN_LOCK ActualLock;
    PVOID DispatchAddress;                //中间的 Dispatch ISR 的函数地址
    ULONG Vector;                         //中断向量号
    KIRQL Irql;                           //ISR 对应的中断优先级
    KIRQL SynchronizeIrql;
    BOOLEAN FloatingSave;
    BOOLEAN Connected;                    //本中断对象是否已经被连接到 IDT 中
    CHAR Number;                          //要关联到哪个 CPU 上
    UCHAR ShareVector;
    KINTERRUPT_MODE Mode;                 //中断模式:Latched(电平触发),LevelSensitive(边
                                          //沿触发)
    ULONG ServiceCount;
    ULONG DispatchCount;
    ULONG DispatchCode[106];              //中断序言代码
} KINTERRUPT, * PKINTERRUPT;
```

这里要强调的是,如果是单 CPU 或者是多处理器体系的第一个 CPU,则对应的数据结构是 IO_INTERRUPT,它内嵌一个 KINTERRUPT 结构,因此本质上系统中首个 CPU 也是与 KINTERRUPT 对应的。当然,除了首个 CPU 的 KINTERRUPT 结构,IO_INTERRUPT 还包括了其他 CPU 对应的 KINTERRUPT 结构的指针数组(单个 CPU 的系统中该数组长度为 0)。另外,“一个 KINTERRUPT 对应着一个 CPU”这一说法也不是很准确,CPU 与 IDT 的确是一一对应的,每个 IDTEntry 也对应一个中断向量,但一个中断向量却不止对应一个 KINTERRUPT 结构(KINTERRUPT 只能表示单个向量的中断对象),那怎么办呢?KINTERRUPT 中有一个 InterruptListEntry 域,这是个 LIST_ENTRY 结构,用于将代表同一个中断向量的 KINTERRUPT 连接成一个链表,如图 7-5 中右半部分所示。

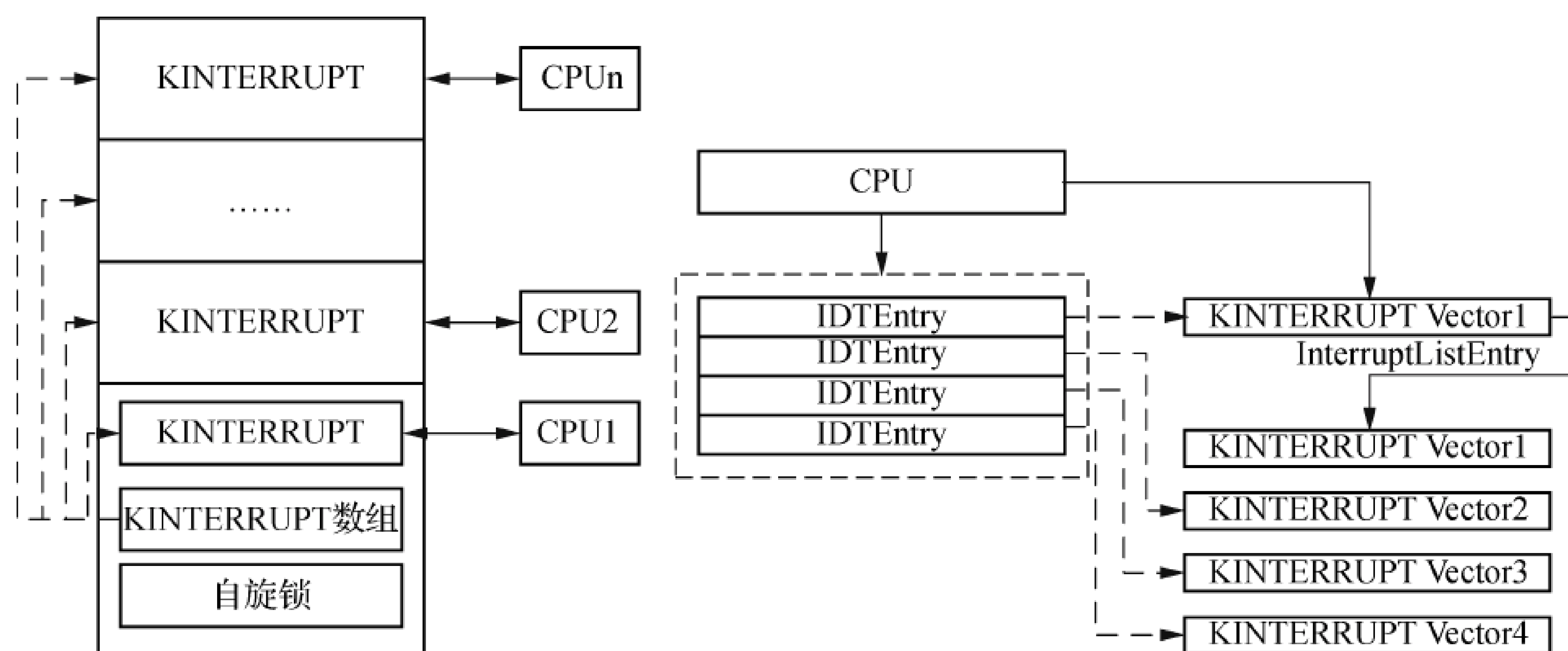


图 7-5　KINTERRUPT 和 IO_INTERRUPT 数据结构示意图

在系统初始化时就要对中断数据结构和机制进行初始化。这个过程中针对每个可以执行中断例程的 CPU 进行两个操作：中断对象 KINTERRUPT 的初始化和中断的连接。

前者的主要工作是对 KINTERRUPT 结构中的各个域进行赋值，值得注意的是最后一个域 DispatchCode，这是个 106 个双字长度的字符数组，用于承载中断服务例程执行的序言部分。提到序言，我们会想到系统调用中从用户态进入内核态时执行的用于构造自陷框架、为一些寄存器赋值的序言代码，这里序言的作用与其比较类似。在初始化 KINTERRUPT 的时候，会将一段模板函数拷贝到 DispatchCode 内存区域，这个模板函数就是 KiInterruptTemplate，这是一段中断执行序言指令的机器码。除了拷贝这一段代码，还要将一个 4 字节的双字（Dword）替换为对应 KINTERRUPT 的地址，即将图 7-6 中的 0 的位置（黑体）赋值为对应 KINTERRUPT 的地址，并使 EDI 寄存器指向该地址。

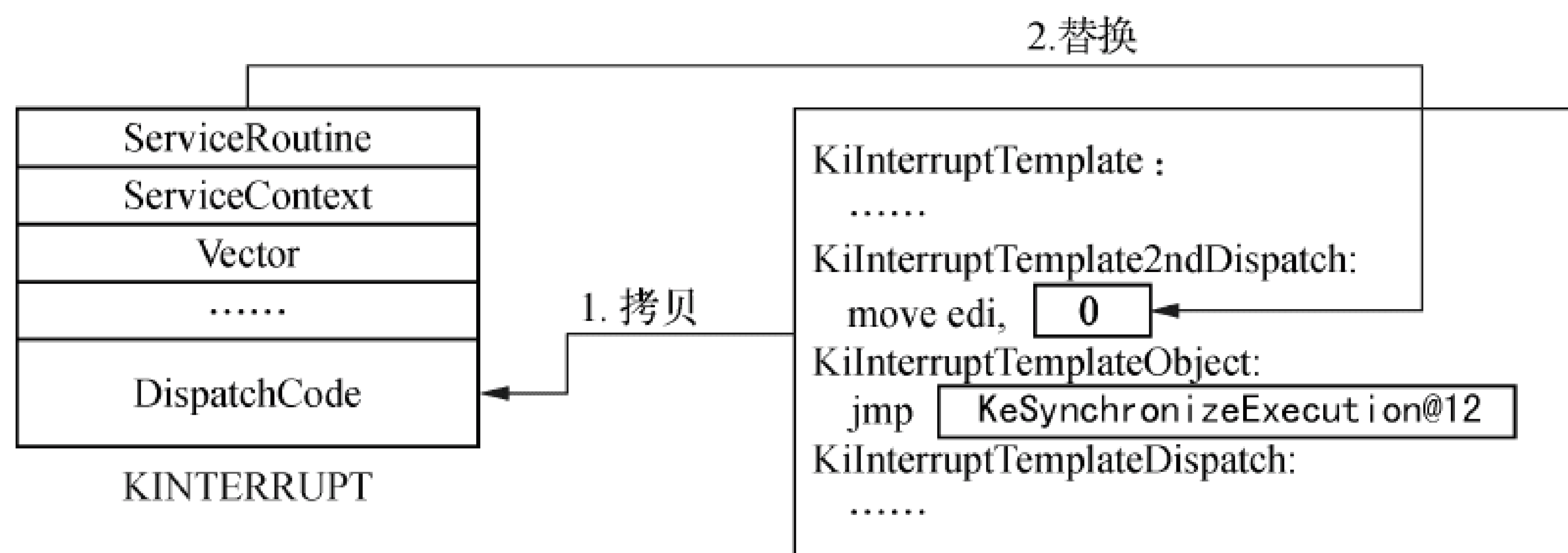


图 7-6　KiInterruptTemplate 代码段与 KINTERRUPT 结构

完成了上述工作，中断结构的初始化工作也就完成了，接下来要进行中断向量的连接，也就是将其挂接到目标 CPU 事先定义的中断向量号上。系统函数 KeConnectInterrupt 用来完成中断连接，其入参是要连接的 KINTERRUPT 结构指针。图 7-7 是 KeConnectInterrupt 的执行流程。

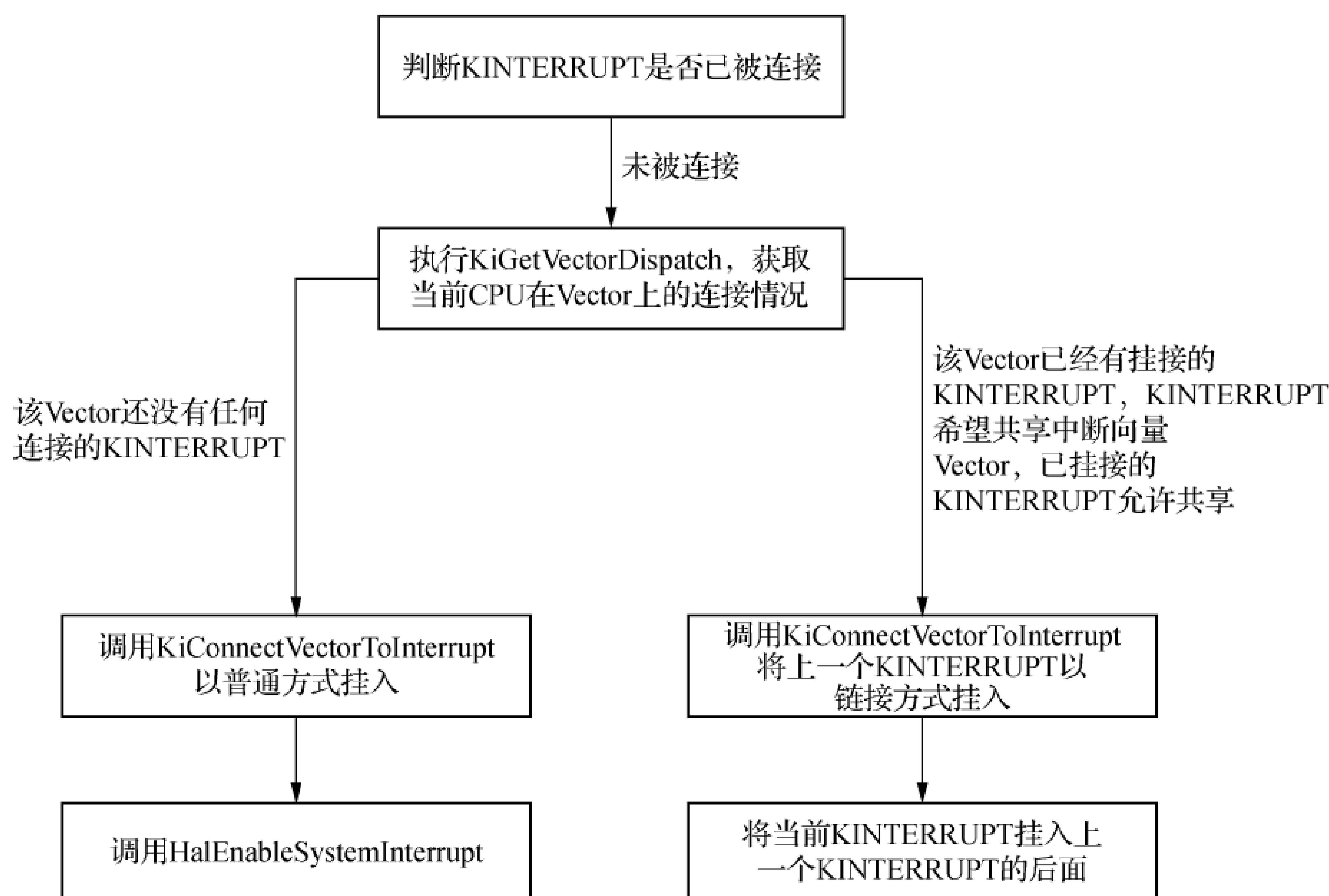


图 7－7　KeConnectInterrupt 的执行流程

首先要判断入参 KINTERRUPT 是否已经被连接。如果已经被连接，现在又要执行一次，这只能说明出错了，这里不具体讨论；如果没有被连接，那么故事还有下文：获取当前 CPU 在 KINTERRUPT 对应的中断向量 Vector 上的连接情况，这是通过 KiGetVectorDispatch 函数来实现的。

KiGetVectorDispatch 通过获取中断向量在 IDT 上的 ISR 入口反推出对应的 KINTERRUPT 结构，从而获取连接信息。获取的连接情况分为两种：

➢ 该 Vector 上还没有任何连接的 KINTERRUPT；

➢ 该 Vector 上已经有了连接的 KINTERRUPT，而入参 KINTERRUPT 又想共享这个中断向量，并且已经存在的 KINTERRUPT 也允许共享，即我们常说的“两厢情愿”。

针对第一种情况，我们调用 KiConnectVectorToInterrupt 以普通方式挂入，并且调用 HalEnableSystemInterrupt 对 PIC 进行工作模式的设置。

针对第二种情况，我们依然调用 KiConnectVectorToInterrupt，但是以链接方式将上一个 KINTERRUPT 重新挂入（第一次挂入这个 KINTERRUPT 时也没想到后面会有其他 KINTERRUPT 与之共享中断向量，因此首次是以普通方式挂入）。之后将当前入参的这个 KINTERRUPT 挂到上一个 KINTERRUPT 的后面。

KiConnectVectorToInterrupt 是 KINTERRUPT 连接的实际操作者，其工作主要分为两步，我们来看看其具体流程（如图 7－8 所示）：

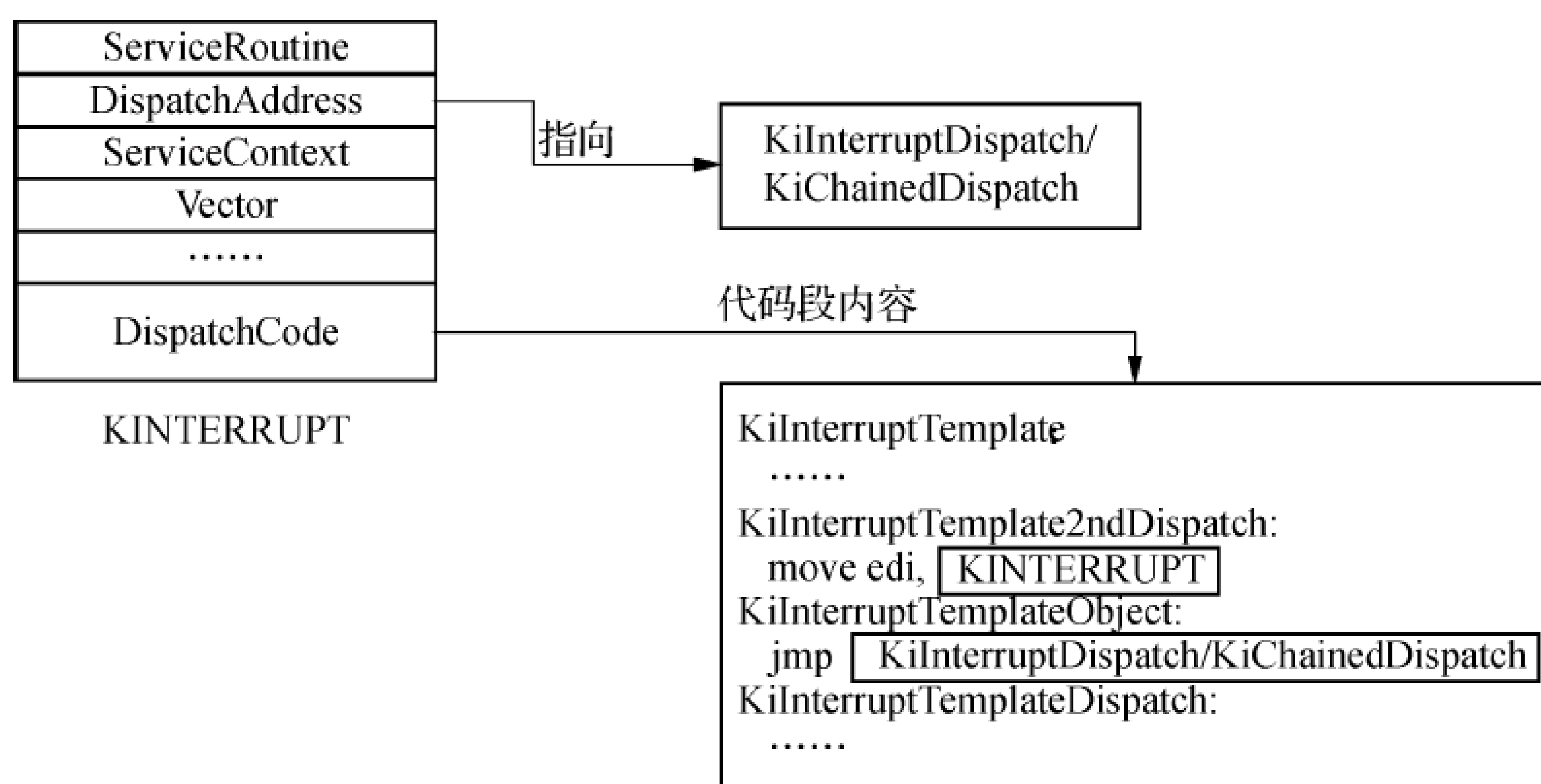


图 7－8　KiConnectVectorToInterrupt 对于 KINTERRUPT 的操作

第一步，完成 KINTERRUPT 的最终修改，关键之处就是使 DispatchCode 中的长跳转指向中断的公共入口：**KiInterruptDispatch** 或 **KiChainedDispatch**。在前文中我们描述过，KINTERRUPT 初始化的时候在 DispatchCode 中替换了一个双字，这个双字就是 KINTERRUPT 的地址，并使 EDI 寄存器指向这个地址，现在又替换了一个长跳转的地址，这预示这个长跳转可能会以 EDI 寄存器的内容为参数（实际上 EDI 寄存器指向中断对象 KINTERRUPT）。而 DispatchCode 也是这个中断向量的响应程序入口代码，因此 DispatchCode 理应作为 IDT 中 IDTEntry 的 ISR 入口指令。故而接下来的第二步就要对 IDT 进行操作和赋值。

第二步，DispatchCode 的基址被赋予 IDT 中对应中断向量 Vector 的中断例程地址中，即 IDTEntry 的 ISR 位置。

中断发生后，在硬件部分走完了流程就来到软件部分（如图 7－9 所示），具体如下：

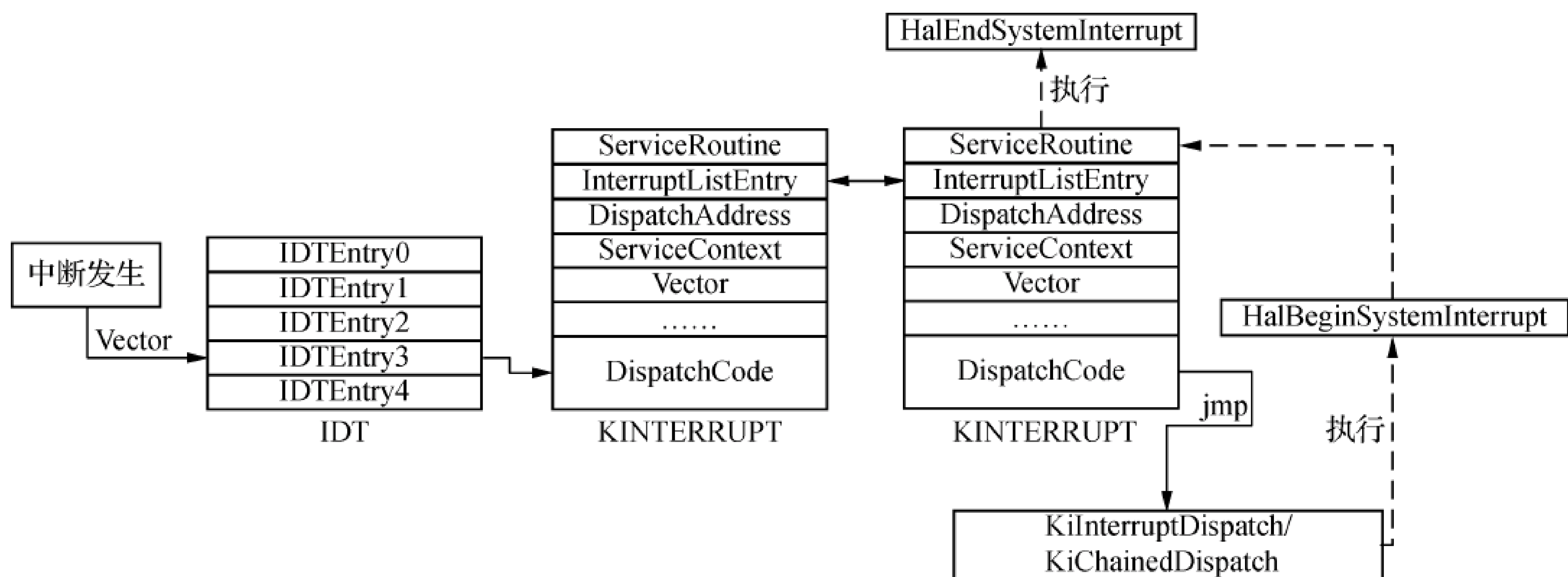


图 7－9　中断处理流程的软件部分

（1）首先会根据当前 CPU 运行状态保存现场。

➢ 如果处于用户态则要为切换堆栈做准备：从 TSS 获取新段描述符，现场保存等；

➢ 如果本来就处于内核态则不需要考虑堆栈切换的情况，只需要保存现场就好。

(2) 然后根据中断向量找到 IDT 中对应的 IDTEntry,进而找到 ISR 的基址,这个基址自然就是目标 Vector 的 KINTERRUPT 的 DispatchCode 基址。

(3) 执行基址中的指令,像系统调用或线程切换一样保存当前现场,最后跳转到 KiInterruptDispatch 或 KiChainedDispatch 函数。

需要注意的是,最后挂入目标 Vector 的 KINTERRUPT 的 DispatchCode 会覆盖其前面的 KINTERRUPT 的 DispatchCode,因此 IDT 中的 ISR 以最后一次挂入的 KINTERRUPT. DispatchCode 为准。

- 以**普通方式**挂入的 KINTERRUPT 执行的是 KiInterruptDispatch,核心操作就是调用传入的 KINTERRUPT 参数中的 ServiceRoutine,而这个 ServiceRoutine 的参数就是 EDI 寄存器中的内容(KINTERRUPT 基址)和 ServiceContext。
- 以**链接方式**挂入的 KINTERRUPT 执行的是 KiChainedDispatch,其步骤稍微复杂一些:以 InterruptListEntry 为链,查找每一个 KINTERRUPT 元素,逐个执行它们的 ServiceRoutine,并且也是以 KINTERRUPT 基址和 ServiceContext 为参数。

不过 KiInterruptDispatch 和 KiChainedDispatch 在执行 ServiceRoutine 的前后要分别执行一下硬件抽象层函数 HalBeginSystemInterrupt 和 HalEndSystemInterrupt:

- **HalBeginSystemInterrupt** 的作用是比较当前中断对应的 IRQL 与当前 CPU 所处的 IRQL,如果大于 CPU 的 IRQL,则表示来了一个优先级更高的中断,需要马上处理;否则,表示中断的 IRQL 尚不足以撼动 CPU 正在执行的线程,所以先将本次中断记录下来待日后处理,并且设置 PIC 屏蔽这一级中断。
- **HalEndSystemInterrupt** 的作用是降低 CPU 的 IRQL。因为 HalEndSystemInterrupt 一定是在 ServiceRoutine 之后才执行,而 ServiceRoutine 是中断的处理函数,在其运行时 CPU 的 IRQL 自然会较高,那么 HalEndSystemInterrupt 在降低 CPU 的 IRQL 后也会检查一下 KPCR 中是否有之前缓存的中断,并且在中断的 IRQL 高于当前 CPU 的 IRQL 时派遣这些中断。

7.2 延迟过程调用机制

中断软件处理流程中还有两个没有提及的重要操作就是关中断与开中断。

在中断处理流程中有的步骤是不能被打断的,要"一气呵成"地完成,因此必须关中断,等"一气呵成"后再开中断,这期间即使有更高级的中断进来也不允许打扰执行。但在中断请求较多的系统中,如果关中断时间过久会严重影响其他中断的执行进而降低整个操作系统的响应效率,因此这个"一气呵成"的过程必须非常短,只处理非常紧急且不得不关中断才能处理的急迫任务,剩下的不那么急迫的、可以开中断执行的任务可以先放一放,"将来的事情将来再说"。自然地,这种一紧一松的操作就把中断分成了上半段和下半段。

一般在 IDT 的 ISR 中处理的都是中断的上半段,上半段处理一些紧急的操作(例如通知

数据到来等)，同时也要为下半段的执行构造上下文框架和环境；而下半段可以在退出 ISR 后的任意线程上下文中执行。这种机制引出了延迟过程调用(Deferred Procedure Call，DPC)的概念。

7.2.1　DPC 的数据结构

在 Windows 中，每个处理器的 KPRCB 有两个 DPC 队列。每当处理完 ISR 的时候，处理器的 IRQL 降到 DISPATCH_LEVEL，便会触发执行 DPC。我们先来看 KPRCB 中与 DPC 有关的域，如下所示：

```
typedef struct_KPRCB
{
    ULONG DpcTime;
    ULONG DpcTimeCount;
    ULONG DpcTimeLimit;
    KDPC_DATA DpcData[2];          //DPC 有两个队列:常规化 DPC 队列和线程化 DPC 队列
    PVOID DpcStack;                //DPC 为防止堆栈溢出,不能使用内核堆栈而需要使用自己的堆栈
    LONG MaximumDpcQueueDepth;
    ULONG DpcRequestRate;
    ULONG MinimumDpcRate;
    UCHAR DpcInterruptRequested;
    UCHAR DpcThreadRequested;
    UCHAR DpcRoutineActive;
    UCHAR DpcThreadActive;
    ULONG DpcLastCount;
    KEVENT DpcEvent;
    UCHAR ThreadDpcEnable;
    LONG DpcSetEventRequest;
    KDPC CallDpc;
} KPRCB, * PKPRCB;
```

其中，DpcData 是个 KDPC_DATA 结构体，与 APC_STATE 非常类似，队列链表头在结构体内部，其具体形态是下面这样的：

```
typedef struct_KDPC_DATA
{
    LIST_ENTRY DpcListHead;      //DPC 队列
    ULONG DpcLock;
    LONG DpcQueueDepth;
    ULONG DpcCount;
} KDPC_DATA, * PKDPC_DATA;
```

KDPC_DATA 中的 DpcListHead 用来挂入 KDPC 数据结构，而 KDPC 对象的结构如下所示：

```
typedef struct_KDPC {
    UCHAR Type;               //表示 DPC 是常规化的还是线程化的,DpcObject 或 ThreadedDpcObject
    UCHAR Importance;                        //紧急程度,以决定是挂在队头还是队尾
    USHORT Number;                           //希望挂入的 CPUID
    LIST_ENTRY DpcListEntry;                 //用于挂入 KDPC_DATA 的 List 中
    PKDEFERRED_ROUTINE DeferredRoutine;      //DPC 的具体执行函数指针
    PVOID DeferredContext;                   //执行 DPC 函数时的上下文环境
    PVOID SystemArgument1;                   //执行 DPC 函数时的参数
    PVOID SystemArgument2;                   //执行 DPC 函数时的参数
    __volatile PVOID DpcData;                //指向所挂入的 KDPC_DATA
} KDPC, * PKDPC, * PRKDPC;
```

7.2.2 DPC 的入队流程

综上所述，每个 CPU 对应的 KPRCB 中含有一个 KDPC_DATA 结构体，每个 KDPC_DATA 结构体包含两个 DPC 队列（常规 DPC 队列和线程 DPC 队列），KDPC 对象被挂入这两个队列中的一个。KDPC 的初始化是通过系统函数 KeInitializeDpc/KeInitializeThreadedDpc 实现的（前者用来初始化常规化 DPC，后者用来初始化线程化 DPC），而挂入 DPC 队列则统一由 KeInsertQueueDpc 实现。

如图 7－10 所示，KeInsertQueueDpc 没有什么门槛，其中 HalRequestSoftwareInterrupt（DISPATCH_LEVEL）就是扫描 DPC 队列的函数，只是将 KPCR 的扩展结构 KPRCB 中的 HalReserved[HAL_DPC_REQUEST]域设置为 true，并不真正执行 DPC 函数，DPC 函数的执行是等到中断优先级下降到 DPC_LEVEL 的时候才进行的。

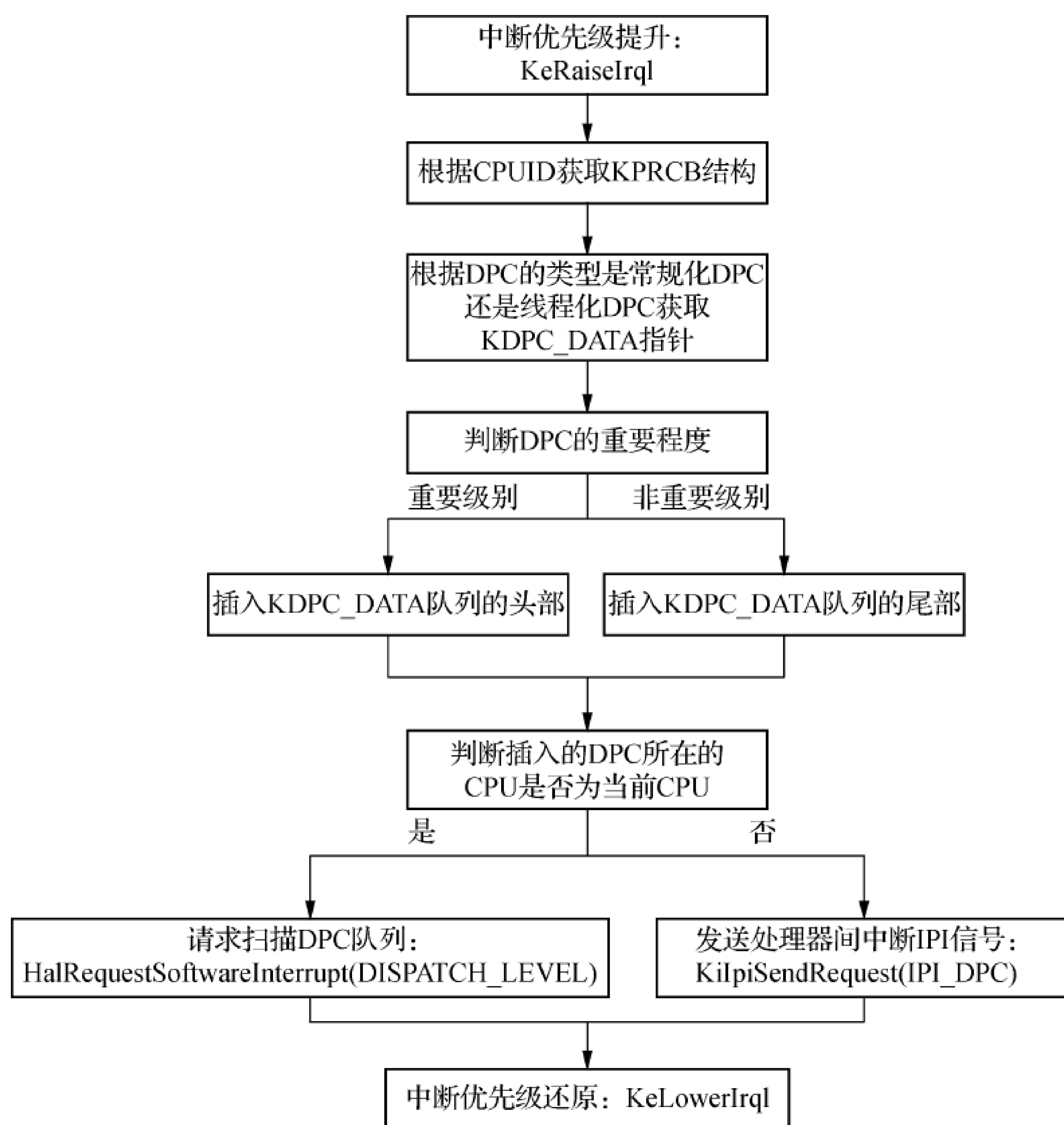


图 7－10　KeInsertQueueDpc 的执行过程

另外，在 DPC 执行过程中是不能发生线程切换的，Windows 通过提升 IRQL 的机制保证了这一点：线程切换时 CPU 的 IRQL 是 DISPATCH_LEVEL，DPC 执行时也是这个级别，因为是同级，因而线程切换无法剥夺 DPC 的执行权。

在系统初始化(KeInitSystem)的时候,如果线程化 DPC 是启用状态的,则除了执行 DPC 表头初始化和设置深度初值之外,还执行以下操作:

➢ 初始化 KPRCB 中的 DpcEvent 为无信号状态的同步事件;

➢ 为每个 CPU 内核创建一个 DPC 线程,以便于执行 DPC。

线程化 DPC 是一种特殊的 DPC,用于 Windows Vista 及之后的版本,我们在此先搁置不谈,因此本节描述的 DPC 都是常规 DPC。

7.2.3　DPC 的执行流程

DPC 是 IRQL 从 DISPATCH_LEVEL 以上降到 DISPATCH_LEVEL 或以下级别的时候才执行的,我们以 KeLowerIrql 为例来讲解 DPC 的执行流程。KeLowerIrql 的执行过程如图 7-11 所示。

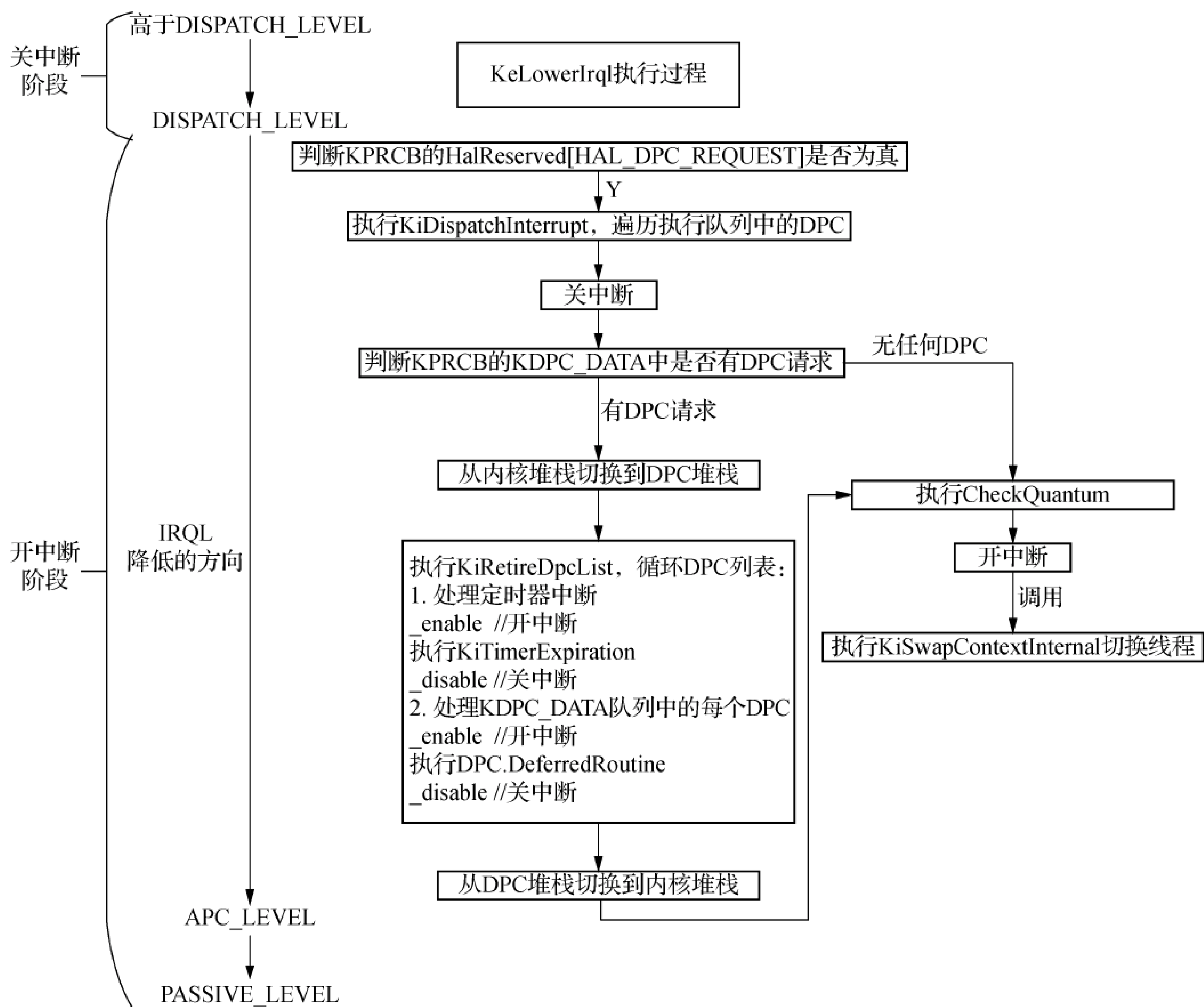


图 7-11　KeLowerIrql 的执行过程

在 KeLowerIrql 的执行过程中,处理器的 IRQL 从处理中断时的 DISPATCH_LEVEL 级别以上降到 PASSIVE_LEVEL,中间经历了 DPC 遍历执行、线程切换、APC 投递执行和普通线

程运行等 IRQL 不同的几个阶段。其中在 DISPATCH_LEVEL 级别上要执行以下操作：

（1）判断处理器 KPRCB 中的 HalReserved[HAL_DPC_REQUEST]是否为真，为真则表示 PRCB 中有 DPC 等待执行；为假则表示没有 DPC 等待，此时 IRQL 可以降到 APC_LEVEL。

（2）如果为真，则执行 KiDispatchInterrupt——预备执行 DPC。首先关中断，防止高级别中断打扰目前的指令执行，然后判断 DPC 的井深（DPC 数量）和定时器中断 DPC 是否不为 0 或已存在，如果两个条件都满足，则：

- 从内核堆栈切换到 DPC 堆栈。当前线程运行在内核态，其堆栈自然是内核堆栈，但是内核堆栈是比较小的，为了防止堆栈溢出，DPC 有专门的堆栈，此时需要将内核堆栈的指针寄存于 DPC 专有堆栈中；
- 执行 KiRetireDpcList，即循环执行列表中的 DPC，包括定时器中断 DPC 和普通 DPC，注意每次执行 DPC 时都要先开中断，执行完再关中断；
- 执行完全部 DPC 后从 DPC 专有堆栈切换回内核堆栈。

（3）执行 CheckQuantum：开中断，执行线程切换。

其实，DPC 的执行本身没有什么技术门槛，只是它是在 IRQL 下降的过程中，具体来说是从 DISPATCH_LEVEL 降到 APC_LEVEL 的过程中发生的，中间要经历堆栈切换和开关中断，最后还要执行线程切换，这一系列操作执行下来就显得比较麻烦了。

本章小结

系统中断是包括 Windows 在内的操作系统处理外围设备信号的机制。例如我们常说的“敲击键盘或点击鼠标时系统会发生什么”这类问题都是由中断机制和视窗型报文机制来诠释的。本章首先讲述中断的硬件和软件处理机制，这分别对应了设备中断发生后从硬件转移到软件响应的过程。

但是，中断的处理往往是耗时耗力的，如果采用同步的方式处理很可能造成系统的卡顿和其他中断信号的丢失，因此将中断的处理分为前后两段，即前半段同步式处理，后半段延迟式异步处理便成了一个可行的选择。Windows 系统支持这种分段，并将后半段命名为延迟过程调用（DPC）。

本章也讲述了 DPC 相关的数据结构、后半段任务的封装和入队流程以及 DPC 的执行过程。这里要注意的是 DPC 拥有自己的堆栈而不占用系统本身定义的堆栈。

第 8 章　视窗型报文

我们常说 Windows 是采用消息驱动机制的，这里的"消息"就是我们接下来要说的视窗型报文。在系统中我们点击鼠标或敲击键盘都会触发视窗型报文，因为这些动作的解释执行必然与操作系统桌面某个区域（视窗）有关。比如，我们用鼠标点击 MFC（Microsoft Foundation Class，微软基础类）程序的某个 Button，当鼠标触发"按下"动作的时候，鼠标驱动程序将"按下"操作翻译成对应的报文，这个报文携带"按下"事件发生时鼠标光标在操作系统桌面上的坐标等信息。内核收到这个报文后寻找光标的坐标对应的视窗（Button 所在的视窗），并将该报文"投递"到这个视窗对应的线程，该线程处理"按下 Button"报文事件，将该报文翻译成 MFC 可以识别的 BN_CLICKED 消息，继而触发 MFC 对应的事件处理，调用注册的处理函数。在整个过程中，视窗型报文驱动着设备驱动模块、内核、应用进程的线程和视窗之间的协作。



图 8－1　本章提纲

本章将按照图 8－1 所示的提纲进行介绍，详细阐述视窗型报文的生产和消费过程，但在此之前要先介绍一些基础知识和技术。

8.1　视窗型报文的数据结构

在 Windows 中，视窗型应用程序的执行就是一个不断获取和处理报文的循环过程，其基本的处理框架如图 8－2 所示。这个循环直到收到 WM_QUIT 消息才会结束，但在收到 WM_QUIT 之前，如果报文队列为空则会阻塞。而报文处理的本质就是调用视窗关联的函数，就拿前面的例子来说，最终还是要调用 MFC 为单击按钮注册的 OnButton 函数。

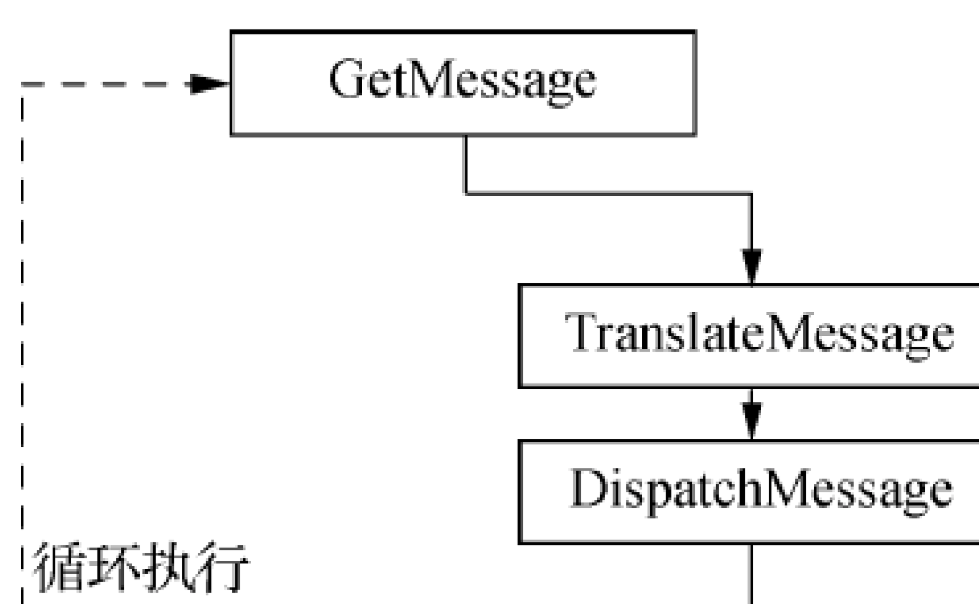


图 8－2　获取报文和处理报文的循环逻辑

在线程的 KTHREAD 结构中，ServiceTable 域指向系统服务描述符表。这个描述符表或者是原生的 KeServiceDescriptorTable，或者是扩展的 KeServiceDescriptorTableShadow。前文讲过，前者只包含索引在 0x1000 以下的系统调用；而后者则包含了索引在 0x1000 以下和 0x1000 以上的系统调用。Windows 将系统调用以 0x1000 为界分为了两部分：原生的系统调用和扩展的系统调用。所谓原生的系统调用就是不涉及视窗绘图操作的调用，如 NtReadFile、NtWriteFile 等，如图 8－3 所示；而扩展的系统调用是涉及视窗绘图操作的 GUI 调用，它们位于 win32k. sys 模块内，如 NtUserGetDC 等，如图 8－4 所示。

序号	函数名称	当前函数地址	是否被挂钩	原始函数地址	当前函数所在的模块
0	NtMapUserPhysicalPagesS...	0xfffff80004a68810			C:\Windows\system32\ntoskrnl.exe
1	NtWaitForSingleObject	0xfffff800048f9ae0			C:\Windows\system32\ntoskrnl.exe
2	NtCallbackReturn	0xfffff800046a7d40			C:\Windows\system32\ntoskrnl.exe
3	NtReadFile	0xfffff800048fd210			C:\Windows\system32\ntoskrnl.exe
4	NtDeviceIoControlFile	0xfffff8000495bdd0			C:\Windows\system32\ntoskrnl.exe
5	NtWriteFile	0xfffff800048fdc20			C:\Windows\system32\ntoskrnl.exe
6	NtRemoveIoCompletion	0xfffff800048fcc50			C:\Windows\system32\ntoskrnl.exe
7	NtReleaseSemaphore	0xfffff800049160d0			C:\Windows\system32\ntoskrnl.exe
8	NtReplyWaitReceivePort	0xfffff800048ee070			C:\Windows\system32\ntoskrnl.exe
9	NtReplyPort	0xfffff80004a3f4c0			C:\Windows\system32\ntoskrnl.exe
10	NtSetInformationThread	0xfffff800048f6860			C:\Windows\system32\ntoskrnl.exe
11	NtSetEvent	0xfffff8000490c86c			C:\Windows\system32\ntoskrnl.exe
12	NtClose	0xfffff800048f9020			C:\Windows\system32\ntoskrnl.exe
13	NtQueryObject	0xfffff800049111d8			C:\Windows\system32\ntoskrnl.exe
14	NtQueryInformationFile	0xfffff80004ac8590			C:\Windows\system32\ntoskrnl.exe
15	NtOpenKey	0xfffff800048eb8a8			C:\Windows\system32\ntoskrnl.exe
16	NtEnumerateValueKey	0xfffff800048f17a0			C:\Windows\system32\ntoskrnl.exe

图 8－3　原生的系统调用描述符表

序号	函数名称	当前函数地址	是否被挂钩	原始函数地址	当前函数所在的模块
99	NtUserInternalGetWindo...	0xfffff96000191bc4	-	0xfffff96000191bc4	C:\Windows\system32\win32k.sys
100	NtUserGetWindowDC	0xfffff96000191438	-	0xfffff96000191438	C:\Windows\system32\win32k.sys
101	NtGdiD3dDrawPrimitives2	0xfffff9600024aa20	-	0xfffff9600024aa20	C:\Windows\system32\win32k.sys
102	NtGdiInvertRgn	0xfffff96000271a90	-	0xfffff96000271a90	C:\Windows\system32\win32k.sys
103	NtGdiGetRgnBox	0xfffff960002bc3b0	-	0xfffff960002bc3b0	C:\Windows\system32\win32k.sys
104	NtGdiGetAndSetDCDword	0xfffff960002bfba4	-	0xfffff960002bfba4	C:\Windows\system32\win32k.sys
105	NtGdiMaskBlt	0xfffff960002baf30	-	0xfffff960002baf30	C:\Windows\system32\win32k.sys
106	NtGdiGetWidthTable	0xfffff9600010dea8	-	0xfffff9600010dea8	C:\Windows\system32\win32k.sys
107	NtUserScrollDC	0xfffff9600012542c	-	0xfffff9600012542c	C:\Windows\system32\win32k.sys
108	NtUserGetObjectInformati...	0xfffff9600016c9c0	-	0xfffff9600016c9c0	C:\Windows\system32\win32k.sys
109	NtGdiCreateBitmap	0xfffff9600016a260	-	0xfffff9600016a260	C:\Windows\system32\win32k.sys
110	NtUserFindWindowEx	0xfffff96000193e20	-	0xfffff96000193e20	C:\Windows\system32\win32k.sys
111	NtGdiPolyPatBlt	0xfffff960002b65f8	-	0xfffff960002b65f8	C:\Windows\system32\win32k.sys
112	NtUserUnhookWindowsH...	0xfffff96000190bb8	-	0xfffff96000190bb8	C:\Windows\system32\win32k.sys
113	NtGdiGetNearestColor	0xfffff9600013016c	-	0xfffff9600013016c	C:\Windows\system32\win32k.sys
114	NtGdiTransformPoints	0xfffff960002bdb9c	-	0xfffff960002bdb9c	C:\Windows\system32\win32k.sys
115	NtGdiGetDCPoint	0xfffff960002bda0c	-	0xfffff960002bda0c	C:\Windows\system32\win32k.sys

图 8－4 扩展的系统调用描述符表

在线程初始化的时候，KTHREAD 的 ServiceTable 域默认指向 KeServiceDescriptorTable 这个原生表，该线程也是个常规非 GUI 线程。但当线程需要调用 0x1000 以上索引的系统服务的时候（视窗操作），ServiceTable 域要重定向到 KeServiceDescriptorTableShadow，且该线程要转变成 GUI 线程，哪怕线程的生命周期中只调用视窗操作一次，这个转换也是必须进行且不可逆的，而这个转换操作是由内核函数 PsConvertToGuiThread 完成的。

既然 KeServiceDescriptorTableShadow 包括了原生的和 GUI 的系统调用，那让 ServiceTable 域在初始化的时候就一步到位地指向它就得了，为何一开始还要指向 KeServiceDescriptorTable 呢？这其实是基于节省系统开销的目的来考虑的。KeServiceDescriptorTableShadow 表中毕竟要使用到 win32k. sys 模块中的函数，虽然这个模块处于系统地址空间，调用的效率也很高，但毕竟要加载包括 win32k. sys 在内的不少系统模块，在以前内存还比较昂贵的时候，能省一点内存就意味着更高的运行效率（可以跑更多的进程），因此 Windows 连这点“博爱之心”也不想给，直接格杀勿论地使 ServiceTable 指向原生表，只有在触发了 GUI 调用这条“红线”的时候才会允许 win32k. sys 的加载，不过这也是“穷家富路”的不得已而为之。

当然，PsConvertToGuiThread 不仅仅转换了 ServiceTable 的指向，也扩充了内核态堆栈的大小。因为视窗线程所用的资源一般比较多，嵌套也比较深，常规大小的内核态堆栈往往不足以支持视窗调用，为了避免堆栈溢出，扩充堆栈大小也是必要的。

同时，PsConvertToGuiThread 也创建了线程所在进程（当前线程变为视窗线程后，所在进程也会变为视窗进程）的 W32PROCESS 结构和当前线程自己的 W32THREAD 结构（对这两个结构前文有过描述，不再赘述）。在 KTHREAD 结构（TCB）中，Win32Thread 域指向的是一个 THREADINFO 结构，只是 THREADINFO 的第一个域是 W32THREAD 结构，因此我们说 Win32Thread 域就指向 W32THREAD 结构也不太准确，但这个问题不大，只要能通过内存的偏移计算出来就行了。当然在 W32THREAD 之外 THREADINFO 还有其他的域，其中一个是 MessageQueue 结构，这是一个指向本线程报文队列的指针，而报文队列的数据结构是 USER_MESSAGE_QUEUE，这是个对本章内容起到灵魂作用的数据结构，也是个包含了 7 种队列的

数据结构，在此我们用图 8－5 来梳理它们之间的关系。

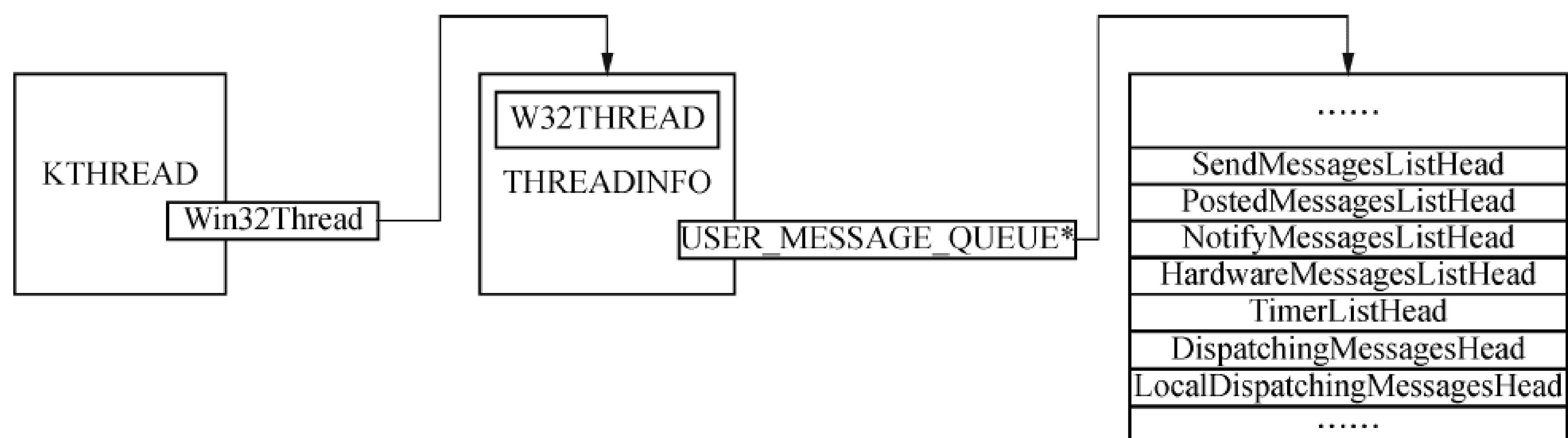


图 8－5　THREADINFO 与 USER_MESSAGE_QUEUE 的结构关系

USER_MESSAGE_QUEUE 数据结构所承载的 7 种队列承载的内容如下所示，下一小节会有更具体的描述：

- **SendMessagesListHead**：发送型报文队列，用于报文接收线程。
- **PostedMessagesListHead**：张贴型报文队列，用于报文接收线程。
- **NotifyMessagesListHead**：通知型报文队列，用于报文接收线程。
- **HardwareMessagesListHead**：来自硬件设备（鼠标、键盘等）的报文队列，用于报文接收线程。
- **TimerListHead**：定时器报文队列，用于报文接收线程。
- **DispatchingMessagesHead**：发送但尚未等到回复的报文队列，用于报文发送线程。
- **LocalDispatchingMessagesHead**：发送型报文暂存队列，用于报文接收线程，在将 SendMessagesListHead 中的报文摘下时先将其暂存在该队列并对发送线程做出回应。

WINDOW_OBJECT 作为视窗对象结构体，其第三个域指向 THREADINFO 数据结构，而 THREADINFO 结构中又包含了 USER_MESSAGE_QUEUE，因此从视窗对象可以关联到具体的报文队列。一个线程只能有一个报文队列，但是一个线程却可以有多个视窗，例如 MFC 进程中的 MAIN_FRAME 和 BUTTON 就是两个视窗，但显然它们要在一个线程中运行。这里要注意，视窗的句柄是全局句柄而非进程内句柄。WINDOW_OBJECT 数据结构如下所示：

```
typedef struct _WINDOW_OBJECT
{
    THRDESKHEAD head;
    PWND Wnd;                                   //视窗句柄
    PTHREADINFO pti;                            //所属线程
    HMENU SystemMenu;                           //左上角的系统菜单
    HWND hSelf;                                 //窗口句柄是内核全局的
    ULONG state;
    HANDLE hrgnUpdate;                          //当前无效区域(指更新区域)的句柄
    HANDLE hrgnClip;                            //剪裁区域的句柄
    struct _WINDOW_OBJECT * spwndChild;         //第一个子窗口
    struct _WINDOW_OBJECT * spwndNext;          //下一个兄弟窗口
    struct _WINDOW_OBJECT * spwndPrev;          //上一个兄弟窗口
    struct _WINDOW_OBJECT * spwndParent;        //父窗口
    struct _WINDOW_OBJECT * spwndOwner;         //拥有者窗口与父窗口是两码事
    PSBINFOEX pSBInfo;                          //滚动条信息
    LIST_ENTRY ThreadListEntry;                 //用来挂入线程的窗口链表
} WINDOW_OBJECT;
```

图 8-6 进一步阐释了视窗对象与线程、报文队列之间的关系，同时也可以看出一个线程可以有多个视窗，每个窗口也都有一个指针指向所属线程的 THREADINFO 结构，而 THREADINFO 则指向线程的视窗型报文队列。由此我们也可以推断出视窗对象不直接关联视窗型报文队列，报文队列只能跟线程关联。接收端线程从报文队列中“挖掘”报文，再根据报文的窗口句柄定位到所属的视窗，进而调用视窗关联的函数。

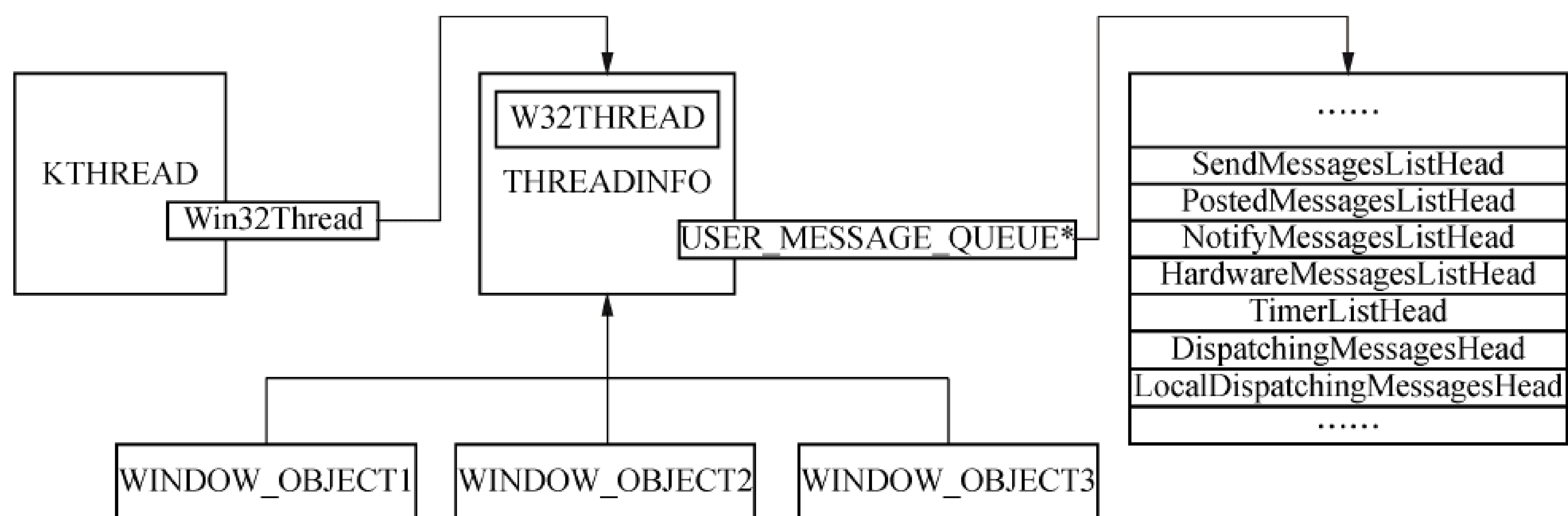


图 8-6　视窗对象与线程、报文队列的关系

明白了视窗和报文的关系，我们接下来看看视窗型报文的接收与发送。

8.2　视窗型报文的接收与发送

我们在前文介绍过，视窗型报文的执行是一个循环，以 GetMessage 为触发和循环条件，以 TranslateMessage 和 DispatchMessage 为执行步骤。其实 TranslateMessage 的主要作用就是处理键盘的扫描码，因此视窗型报文的执行本质上还是 GetMessage 和 DispatchMessage 的循环执行。

GetMessage 的调用核心是 NtUserGetMessage，这是一个 win32k. sys 模块中的扩展系统调用，也是视窗型报文执行的核心步骤：不仅包括了报文的获取，还包括了报文的处理与回复，以及报文获取超时时的阻塞。在介绍 NtUserGetMessage 之前我们先来看看视窗型报文的队列。

前文介绍过，视窗型报文队列分为 7 种类型：发送型报文队列、张贴型报文队列、通知型报文队列、硬件报文队列、定时器报文队列、待回复报文队列和发送型报文暂存队列，那么对应的视窗型报文自然也是 7 种。

我们首先来看发送型视窗报文及其队列 SendMessagesListHead。挂入该队列的报文结构为 USER_SENT_MESSAGE，这个数据结构包含以下几个元素：

➢ **ListEntry**：LIST_ENTRY 结构，用于挂入视窗型报文接收端线程的 SendMessagesListHead 队列，接收线程处理报文时将其从队列中摘下暂存到 LocalDispatchingMessagesHead 中，并准备向发送线程进行回复。

➢ **Msg**：视窗型报文本身。该元素包括视窗的句柄、报文类型、时间标记、光标的坐标





等，还包括一个结构体指针，这个结构体可以由我们自己来定义和分配内存以用作数据扩展，只需将内存地址赋值到这个指针即可。不过“自己来定义”并不是说可以自由指定结构，而是从许可范围内选择结构体类型，包括：WM_CREATE（应用程序调用 CreateWindow 或者 CreateWindowEx 创建视窗时发送此报文）、WM_DDE_ACK、WM_DDE_EXECUTE、WM_GETMINMAXINFO、WM_GETTEXT、WM_SETTEXT 等，这些类型分别针对不同类型的报文，因此其数据结构也是相对固定的。

- **CompletionEvent**：用来唤醒发送端的事件（Event）。
- **SenderQueue**：一个指向 USER_MESSAGE_QUEUE 结构的指针。USER_MESSAGE_QUEUE 包含了 7 种报文队列，SenderQueue 指针指向的是发送端的报文队列。
- **DispatchingListEntry**：LIST_ENTRY 结构，用于挂入发送端的 DispatchingMessagesHead 队列（待回复报文缓存队列），专门等待接收线程对该报文的回复，收到回复后即摘除报文。
- **CompletionCallback**：指向发送端的 Callback 函数，以备发送型报文队列要求调用发送端的回调函数。

8.2.1 视窗型报文的接收

1. 发送型报文的接收

从图 8－7 可以看出，一个发送型报文“一手托两家”：一方面连接发送端线程的发送队列，另一方面连接接收端线程的接收队列，而报文的发送与接收过程与这“两家”紧密相关。因此，发送型报文的发送与接收流程大致如图 8－8 所示。

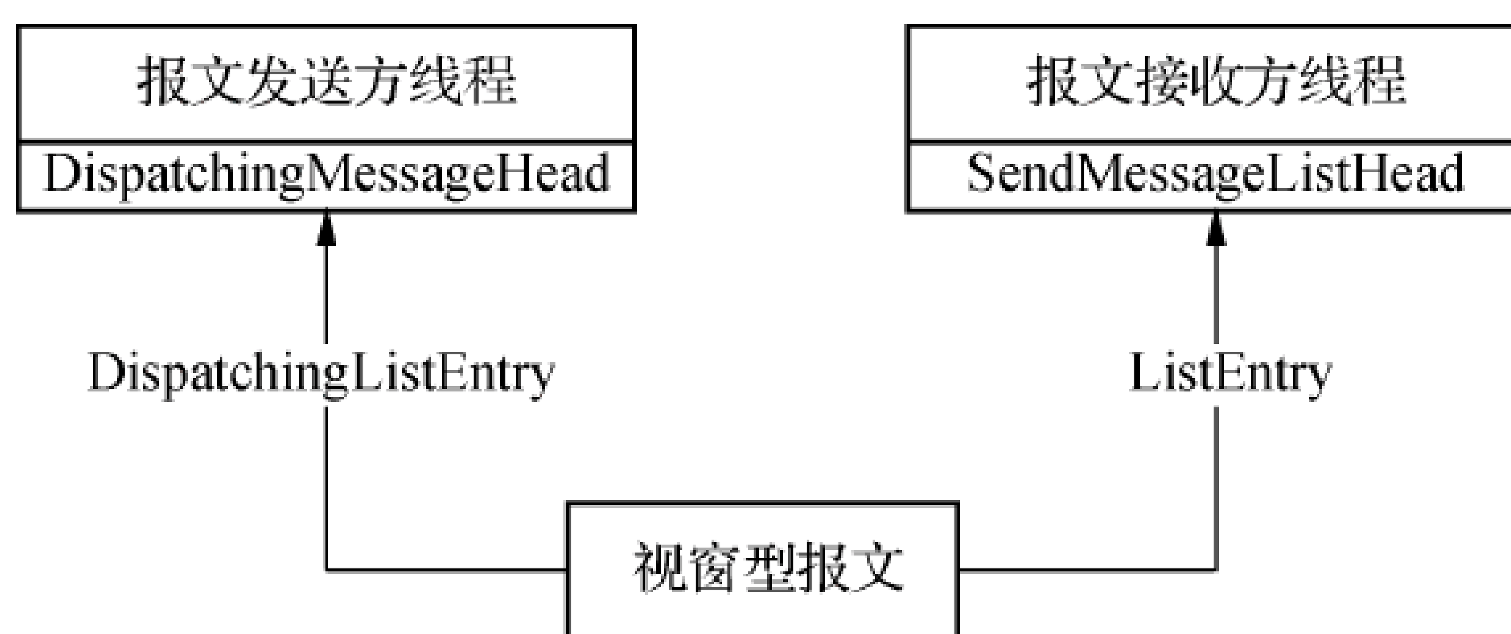


图 8－7 发送型报文示意图

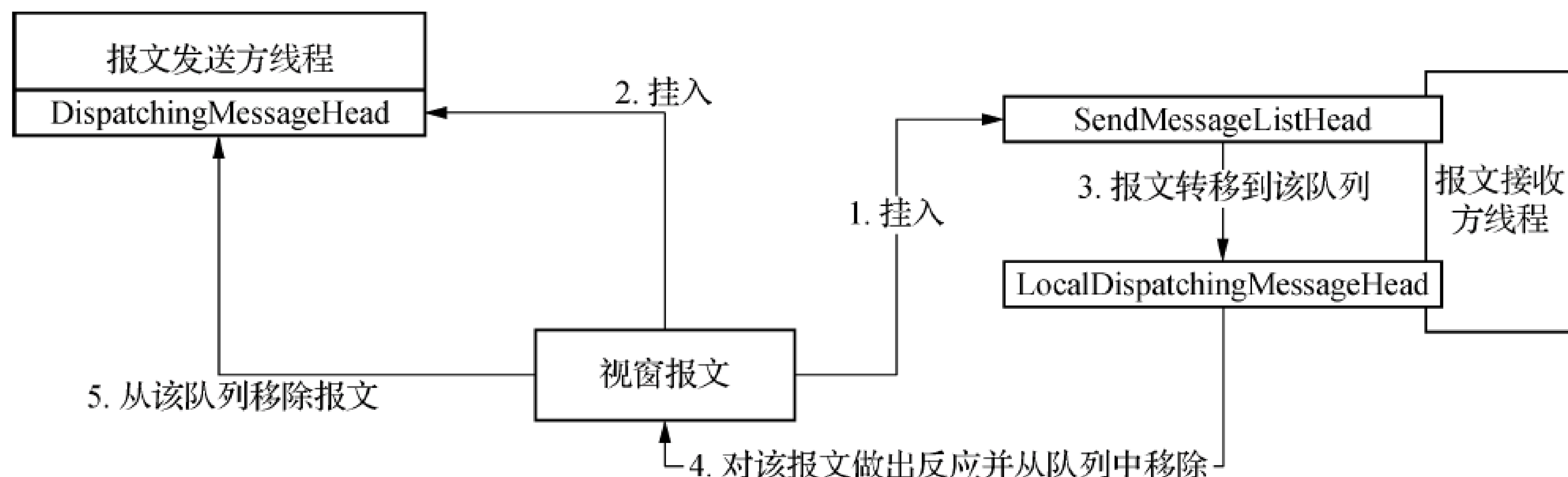


图 8－8 发送型报文的发送与接收流程

从图 8－8 中看出，当发送一个发送型报文时：

（1）发送端线程先将其挂入接收端线程的 SendMessagesListHead 队列，同时也挂入发送端线程的 DispatchingMessagesHead 队列。

（2）在接收端线程的 SendMessagesListHead 队列中循环处理每个 USER_SENT_MESSAGE：先将其从 SendMessagesListHead 队列移除，转而挂入 LocalDispatchingMessagesHead 队列暂存。

（3）对视窗型报文进行处理，也就是调用该报文所关联视窗的处理函数 WndProc，这个函数是用户态的，调用完成后从 LocalDispatchingMessagesHead 队列中移除该报文。

（4）从发送端线程的 DispatchingMessagesHead 队列中彻底移除该报文。

这里我们要简要说明一下。报文是 NtUserGetMessage 从内核中获取的，因此传出 GetMessage 的时候要对上述内核态报文做一定的转化，转化为用户态报文，这样应用进程才能正常使用。NtUserGetMessage 就肩负了这样的任务。NtUserGetMessage 的执行流程如图 8－9 所示。

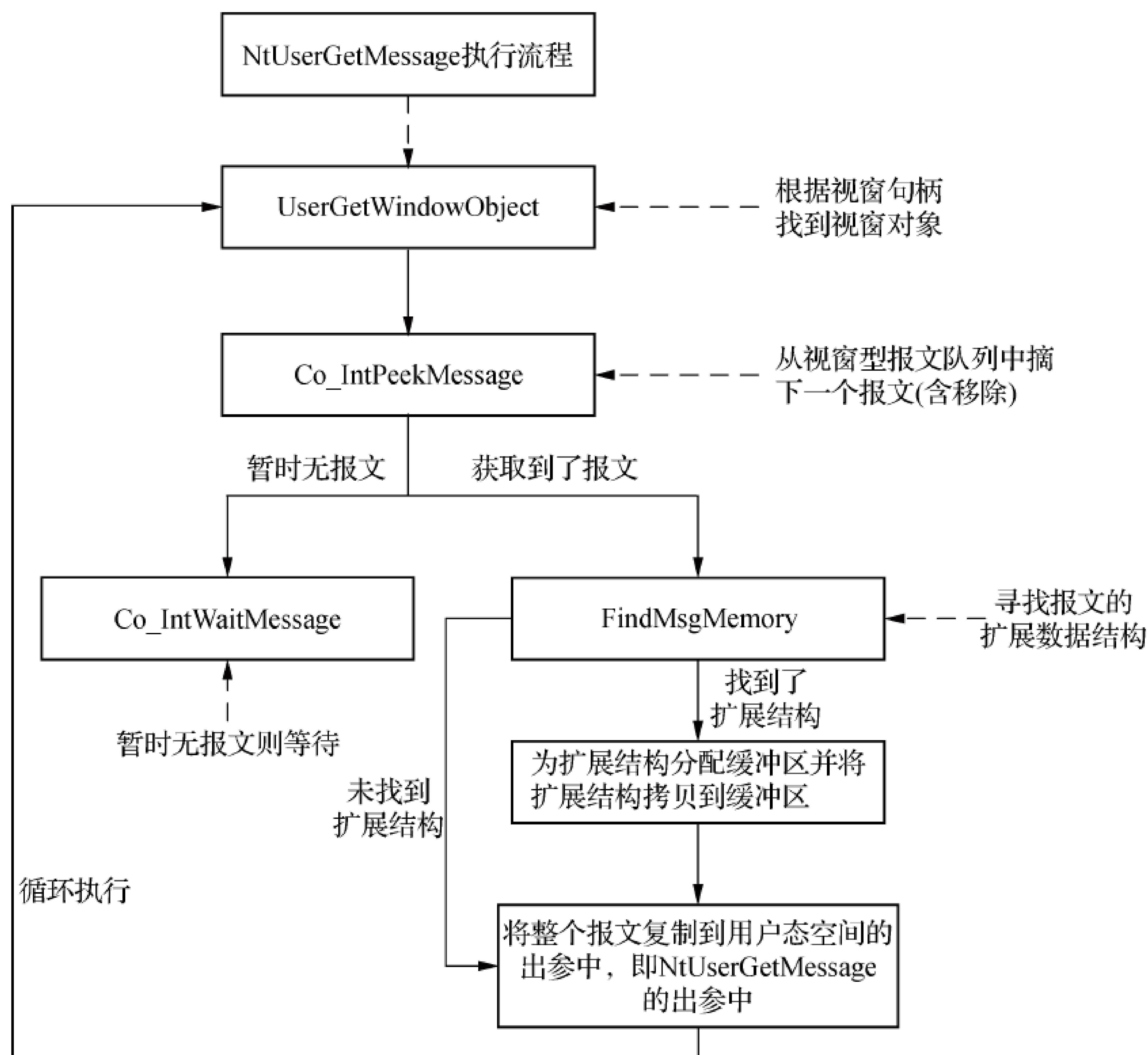


图 8－9　NtUserGetMessage 的执行流程

NtUserGetMessage 的执行分为以下几个步骤：

（1）首先根据窗口句柄获取视窗对象。这是因为报文是与视窗关联的，对报文的处理就是调用视窗关联的函数。

（2）执行循环，通过 Co_IntPeekMessage 从 SendMessagesListHead 中获取一个视窗型报文

并从这个队列中将其摘除。注意,这里是处理发送型报文。

➢ 如果不能从 SendMessagesListHead 中获取报文,则调用 Co_IntWaitMessage 等待;

➢ 若能够获取报文,则探查一下报文是否还有扩展数据结构,因为这个结构在 Msg 中只是一个指针,还需要从该指针所指地址中拷贝出这个扩展数据结构进行判断。

(3) 如果上一步中能获取到报文或者报文所携带的扩展数据结构,就将其一并拷贝到出参中,这个出参是个用户态空间的数据结构(由 NtUserGetMessage 所携带的出参)。

(4) 跳转到步骤(2)循环执行。

由此我们也可以看出,NtUserGetMessage 的核心就是对 Co_IntPeekMessage 的调用,下面我们来展开 Co_IntPeekMessage 的执行流程,如图 8-10 所示。

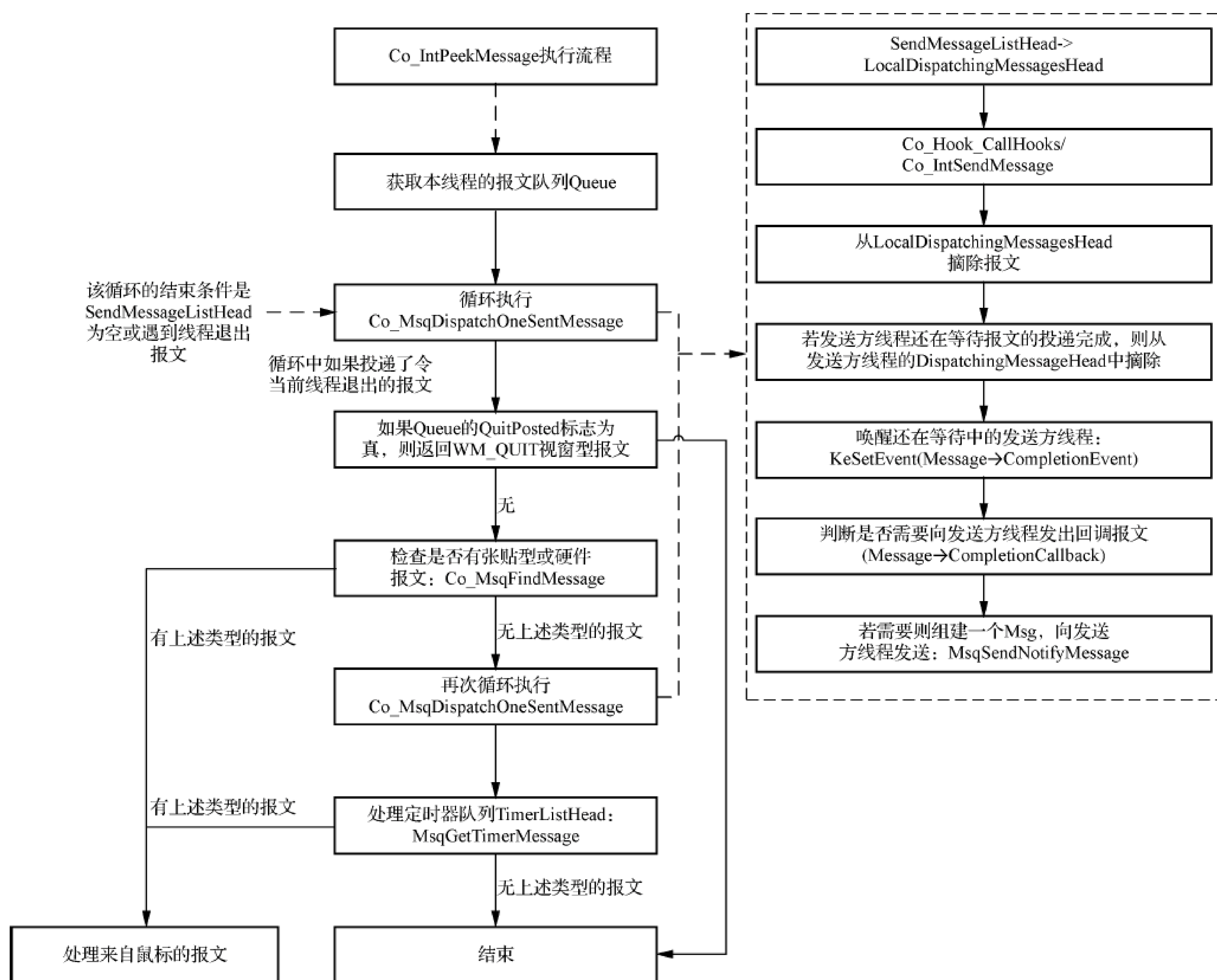


图 8-10 Co_IntPeekMessage 执行流程

Co_IntPeekMessage 首先获取本线程的报文队列指针,即 USER_MESSAGE_QUEUE 结构的指针,这也从侧面印证了报文队列是与线程一对一相关联的,在一个线程中可以有多个视窗,报文队列可以为本线程中的任何视窗服务。然后以刚刚获取的队列结构指针为参数来循环执行 Co_MsqDispatchOneSentMessage。这个函数在队列中处理所有发送型报文,其具体流程参照图 8-10 中的右半部分:

（1）将报文从 SendMessagesListHead 队列转移到 LocalDispatchingMessagesHead 队列。

（2）判断报文是否有挂钩：如果有挂钩，则调用挂钩接口；否则，调用 Co_IntSendMessage，该函数调用视窗的关联函数。SendMessagesListHead 队列是线程的队列，该函数负责从队列中获取所有视窗报文，并根据报文的窗口句柄调用线程所辖窗口（线程中可能有多个视窗）的 WndProc 函数。

（3）从 LocalDispatchingMessagesHead 队列中移除该报文。

（4）如果发送端线程也在等待报文的投递完成，由于此时实质上已经完成了投递，因此也要从发送线程的报文队列 DispatchingMessagesHead 中摘除该报文。

（5）唤醒发送端线程，由于此刻发送端线程很可能还在阻塞等待，既然已经执行完了报文接收端的工作，那么就应该唤醒正在阻塞中的发送端线程，对报文的 CompletionEvent 对象进行激活。

（6）如果还需要调用发送端的回调接口，则对报文的 CompletionCallback 函数进行调用：通过向发送端线程发送通知报文，构造一个报文挂到发送端线程的报文队列 NotifyMessagesListHead 中（通过调用 MsqSendNotifyMessage 接口）来完成函数调用，回调函数的调用者是发送端线程（此时已经结束阻塞等待）。

报文投递的核心操作是执行 Co_IntSendMessage，该函数调用相应视窗的 WndProc 函数，WndProc 函数的原型是下面这样的：

```
LRESULT CALLBACK WndProc(HWND hWnd,UINT message,WPARAM wParam,LPARAM lParam),
```

其中，hWnd 是要处理窗口的句柄；message 是消息 ID，代表了不同的消息类型；wParam 的值为按下按键的虚拟键码；lParam 则存储按键的相关状态信息（比如当鼠标消息发出时，wParam 的值为鼠标按键的信息，而 lParam 则储存鼠标的坐标）。

宏观地看，Co_IntSendMessage 的执行流程是这样的：

（1）针对单个视窗的报文，调用目标视窗的 WndProc 函数 Co_IntCallWindowProc；

（2）针对所有视窗的报文，首先获取桌面视窗，然后通过 IntWinListChildren 获取该桌面视窗的所有子视窗，针对这些子视窗分别调用目标视窗的 WndProc 函数 Co_IntCallWindowProc。

这里的 Co_IntCallWindowProc 涉及在用户态空间执行目标视窗的 WndProc 函数，其具体机制将在下一节详细展开。

执行完上述步骤后，Co_MsqDispatchOneSentMessage 也就完成了一次发送型报文的投递。接下来判断队列中是否有要求线程结束的 QUIT 报文，有则返回一个 WM_QUIT 报文并且结束执行，没有则继续向下执行并判断是否有张贴型报文或硬件报文，没有的话就再次调用 Co_MsqDispatchOneSentMessage 处理 SendMessagesListHead 队列中的发送型报文。

因为在上述的处理过程中很可能当前线程又接收了发送型报文，而这类报文往往要阻塞发送端线程，因此必须优先执行，在这里再次循环调用 Co_MsqDispatchOneSentMessage 也是为了尽快处理这类报文以免长时间的阻塞。最后处理定时器队列中的报文。

2. 张贴型报文的接收

张贴型报文的处理就比发送型报文简单多了,发送端线程只需要将报文挂到目标线程的 PostedMessagesListHead 队列,而不需要等待报文的回复或者被回调,因此也就不存在阻塞等待的环节。同时张贴型报文的数据结构 USER_MESSAGE 也非常简单,除了 Msg 报文结构外,还包括了挂入 PostedMessagesListHead 的 LIST_ENTRY,但是不包含发送端回调函数等这些需要同步交互的元素。将两者类比,张贴型报文好比异步通信,发送型报文好比同步通信。

我们从前文关于发送型报文的一长串描述可以看出:

- GetMessage 的核心操作是执行 NtUserGetMessage;
- NtUserGetMessage 的核心操作是执行 Co_IntPeekMessage(当从发送型报文队列取不到报文时才会执行 Co_IntWaitMessage);
- Co_IntPeekMessage 的核心操作则是执行 Co_MsqDispatchOneSentMessage:这个函数采用 Co_IntSendMessage 投递发送型报文,接下来才处理张贴型报文、硬件报文和定时器报文。

在前文所述的 GetMessage→DispatchMessage 这个最外层循环中,DispatchMessage 的核心操作是执行 NtUserDispatchMessage,这个函数用来调用应用软件通过 RegisterClassEx 向内核注册的 WndProc 窗体事件处理函数(RegisterClassEx 用于告诉进程窗体管理器所注册的窗体属性,如背景色、窗体上的鼠标样式以及窗体事件处理函数等)。NtUserDispatchMessage 在执行过程中会判断是否需要内核处理:例如报文的窗口句柄为空或者句柄有错误而找不到目标窗口,甚至是目标窗口不属于当前线程,这种比较特殊的情况需要内核处理,否则不需要内核处理。如果不需要内核处理,则根据从 NtUserDispatchMessage 返回的 WndProc 函数调用 IntCallWindowProc,也就是先在用户态空间构筑执行环境并切换到用户态,进而执行返回的用户态函数 WndProc。至于怎么在内核态空间构筑用户态空间堆栈并使 WndProc 在用户态空间得到调用就是下一节的内容了。

8.2.2 视窗型报文的发送

下面我们来看视窗型报文的发送。

1. 张贴型报文的发送

对于张贴型报文,Windows API 的 PostMessage 负责消息投递,其调用内核函数 UserPostMessage 针对特定视窗报文、广播型报文和 QUIT 报文分别进行处理:

- 对特定视窗报文要根据句柄找到视窗,并将报文拷贝到内核态空间,继而调用 MsqPostMessage 发送消息报文;
- 针对广播型报文要先获取桌面所有的视窗,而后按照针对特定视窗报文的处理办法来处理;
- 针对 QUIT 报文则直接调用 MsqPostQuitMessage,该函数对目标视窗所在线程的消息

队列做标记并激活相关等待事件，即 MessageQueue 的 NewMessages 标志，表明新的消息到来，以便回标线程处理 QUIT 消息。

因此，UserPostMessage 的核心就是执行 MsqPostMessage。MsqPostMessage 的主要工作是：创建和初始化视窗型报文，将该报文挂入目标线程（或称为目标窗口）的 MessageQueue 的 PostedMessagesListHead 队列，并设置 MessageQueue 的 NewMessages 标志，表明新的消息到来，需要唤醒接收端线程。

2. 发送型报文的发送

对于发送型报文，Windows API 的 SendMessage 负责消息投递。该函数先将报文从用户态空间拷贝到内核态空间，再调用 NtUserSendMessage 发送报文。在这个发送过程中，该函数也会判断目标视窗是当前线程还是其他线程。如果是当前线程，则在当前函数中处理视窗报文即可，否则要将报文挂入其他线程的报文队列并等待这些线程被唤醒。SendMessage 的执行流程如图 8－11 所示。

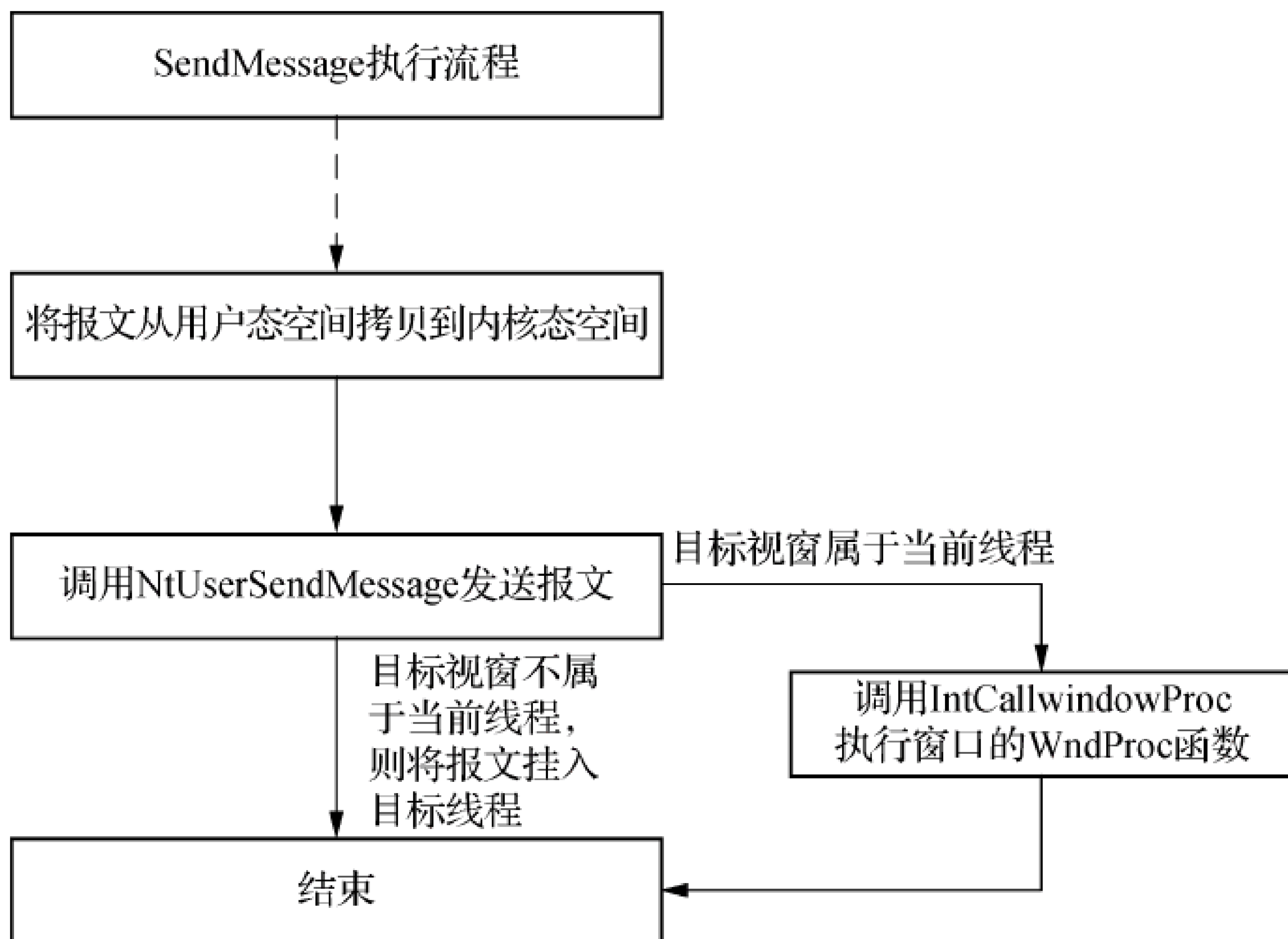


图 8－11　SendMessage 的执行流程

NtUserSendMessage 调用 Co_IntDoSendMessage 进行报文发送。Co_IntDoSendMessage 首先判断句柄是否有效，如果根据入参句柄不能获取到目标视窗，或者目标视窗不属于当前线程，或者报文是广播型报文，那还是要留到内核中去处理的，不过也不必立即处理，也就不必在当前线程调用 IntCallWindowProc，而是调用 Co_IntSendMessage 或 Co_IntSendMessageTimeout 挂入目标窗口所属线程的报文队列；反之，若报文是非广播型报文或者目标视窗属于当前线程就不需要这么一番周折了，直接在 NtUserSendMessage 函数中调用 IntCallWindowProc 来处理即可，这意味着消息的发送和处理都在当前线程，只是要回到用户态空间执行 WndProc 函数。

Co_IntSendMessage 和 Co_IntSendMessageTimeout 并没有本质的不同，对于非当前线程的视窗，我们来看 Co_IntSendMessageTimeout。Co_IntSendMessageTimeout 的逻辑比较简单：

➢ 针对特定视窗报文直接发送到目标视窗(Co_IntSendMessageTimeoutSingle);

➢ 针对桌面视窗则先获取当前桌面下的所有视窗,遍历这些视窗并将该报文发到这些视窗中。而这个发送函数的核心工作又分为两部分:

- 对于目标视窗属于当前线程的情况,使用 Co_IntCallWindowProc 回调视窗的 WndProc 函数,即从内核态空间回调用户态空间的函数;
- 对于目标视窗不属于当前线程的情况,既要将报文挂入本线程的投递队列 DispatchingMessagesHead,也要挂入目标视窗所在线程的 SentMessagesListHead 队列,同时唤醒可能正在等待接收报文的目标线程(置位 MessageQueue 的 NewMessages 标志)。

那么接下来,发送端线程就要睡眠等待了。在等待的过程中,如果接收端线程处理完了报文,则发送端线程会被唤醒,在被唤醒的档口循环执行 Co_MsqDispatchOneSentMessage,这是因为本地可能也会有发送型报文要投递。

发送型报文是非常紧急的,而且会使发送端线程阻塞,快手快脚地处理完发送型报文无论对于接收端还是发送端而言都是明智之举。

8.3 用户态空间回调机制

上节留了一个尾巴,所谓对视窗型报文的处理就是调用视窗的 WndProc 函数,但是 WndProc 都是在用户态空间中的,而报文的处理是在内核态空间中的。那怎样做到在内核态空间回调用户态空间的函数呢?这就是本节要解决的问题。

虽然 APC(异步过程调用)也可以实现从内核态空间向用户态空间的回调,但因为发送型报文类似同步通信机制,不允许在线程切换的档口执行用户态空间部分,因此类似 APC 这样的有些延迟和异步的调用方式并不适用于视窗型报文处理的场景,所以必须有一种新的回调机制来处理堆栈切换和跨空间接口调用等一系列操作。

win32k.sys 有一种特殊的机制,可以用来完成上述场景的堆栈切换和接口调用。虽然这样切换的效率不是很高,但毕竟这种切换机制可以做到同步切换。考虑到视窗型报文也不会发生得很频繁,这种机制应付视窗操作的用户态空间回调还是足够了的。在这里,我们规定了 6 类回调操作,如图 8-12 所示。

在 PEB 的回调函数表域 KernelCallbackTable 中存在 6 个元素,分别对应了 user32.dll 里面的 6 个函数指针(索引值如图 8-12 所示),这些函数用于从内核态空间向用户态空间回调。当一个进程加载 user32.dll 的时候就需要设置好回调函数表。我们在回调用户态空间函数的时候,以索引值为参数,以系统调用 KeUserModeCallback 为统一入口,不同的索引值对应了不同的函数指针,函数所需要传入的参数块也是不一样的。不过无论参数块有多少不同,视窗句柄和视窗的函数指针一定是存在的,KeUserModeCallback 这个函数会构筑用户态堆栈以回调视窗函数。

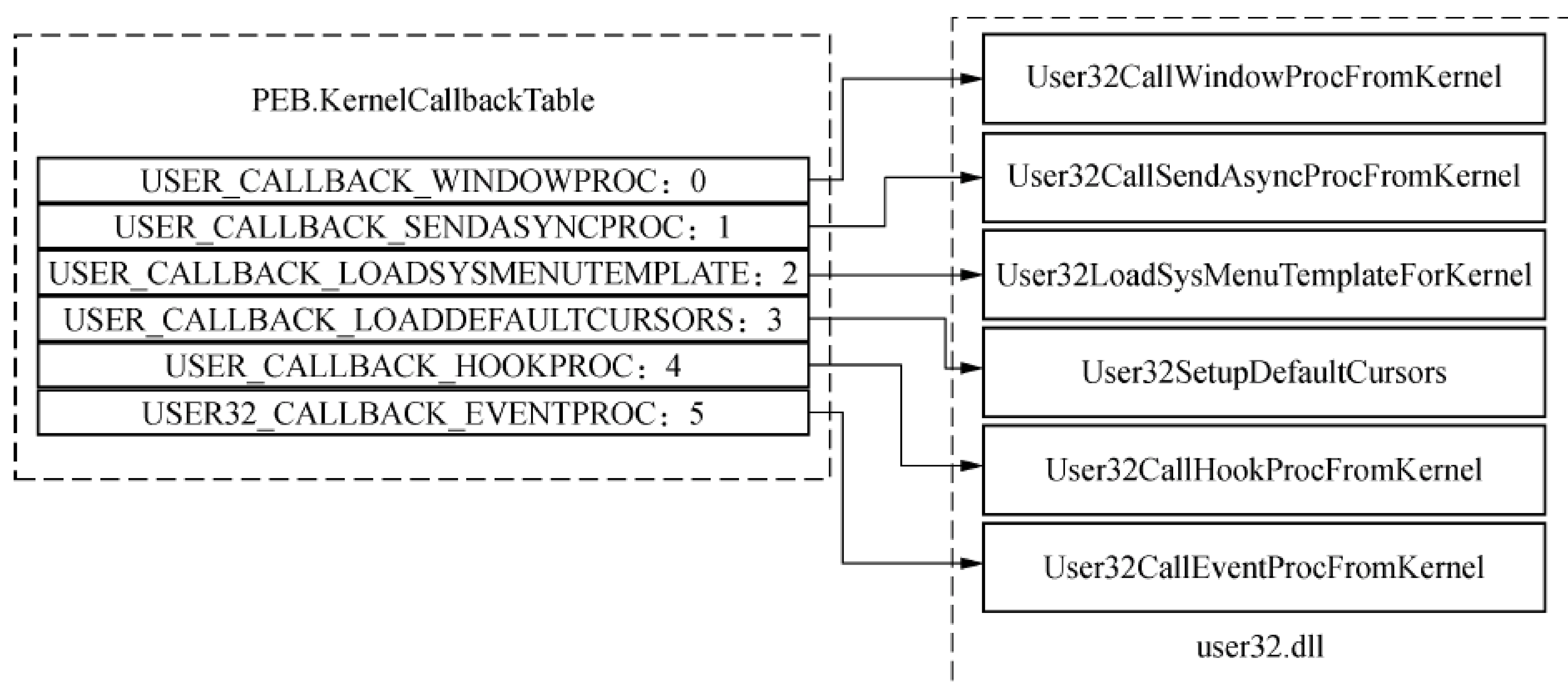


图 8-12　PEB 中的回调函数表

下面介绍在内核态空间执行 KeUserModeCallback 的过程。

首先改变用户态空间堆栈。因为要在用户态空间执行回调函数，所以需要在这里构筑回调函数的调用堆栈，如图 8-13 中左半部分所示。由于改变了用户态空间堆栈地址，因此要将新的堆栈地址更新到内核态空间堆栈原有的自陷框架中，如图 8-13 右半部分所示。

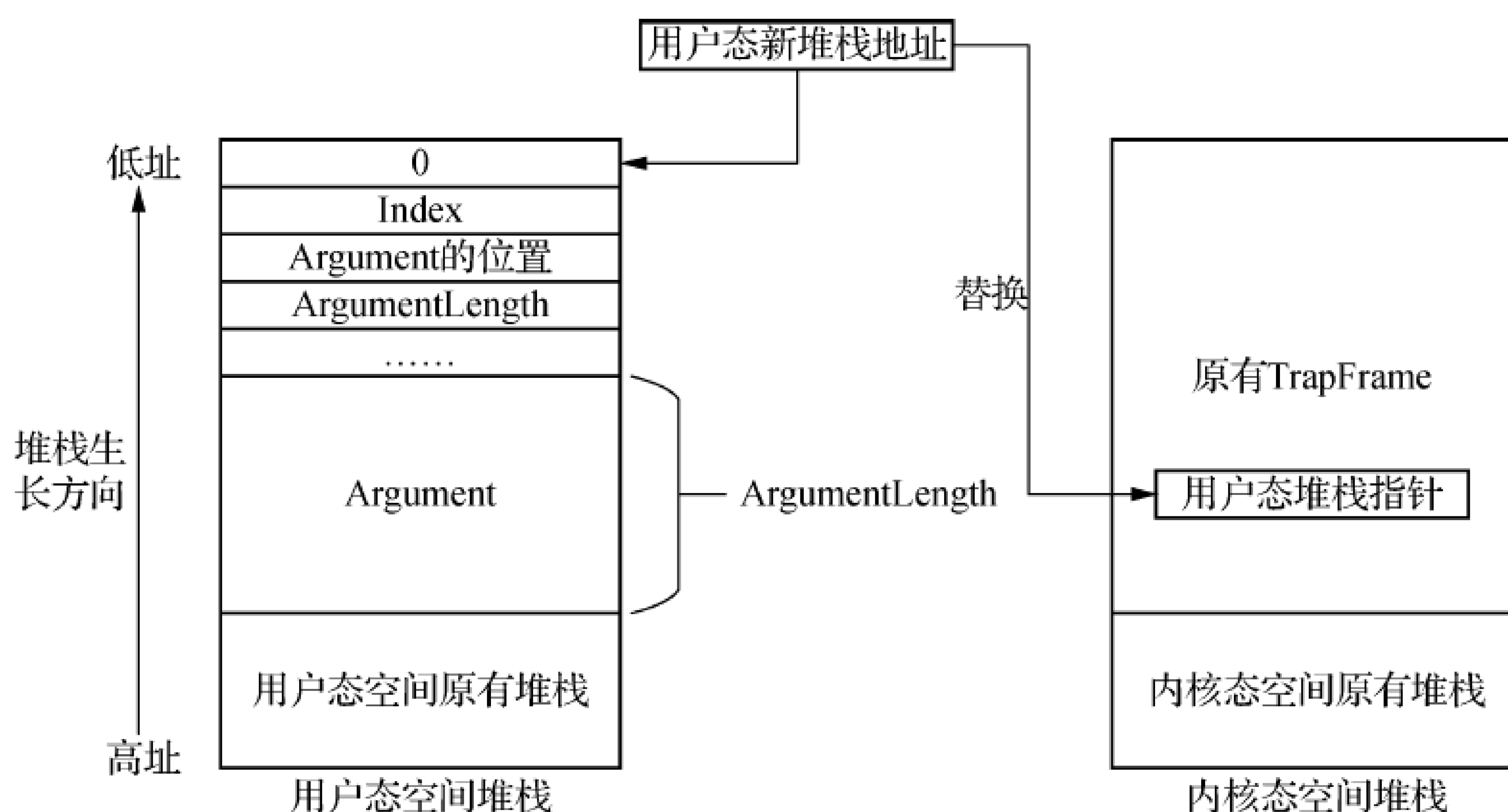


图 8-13　KeUserModeCallback 对于堆栈的修改

堆栈安排好了以后，KeUserModeCallback 调用 KiCallUserMode 继续构造和完善内核堆栈，完成后调用 KiServiceExit 返回用户态空间。KiCallUserMode 对于内核堆栈的修改如图 8-14 所示，具体如下：

- 在原有的内核堆栈上构造 KiCallUserMode 的调用框架，并将参数和返回地址压栈；
- 保存 EBP 等寄存器的值，这里只需要保存 4 个寄存器的值；
- 将原有回调框架的位置、自陷框架指针等压栈，并更新回调框架指针；
- 构筑新的自陷框架和浮点运算框架，其中浮点运算框架可以与内核态空间堆栈中前一个 NPX_FRAME 一致，自陷框架也与前一个 TRAP_FRAME 基本一致，只是 EIP 更改为 KeUserCallbackDispatcher。

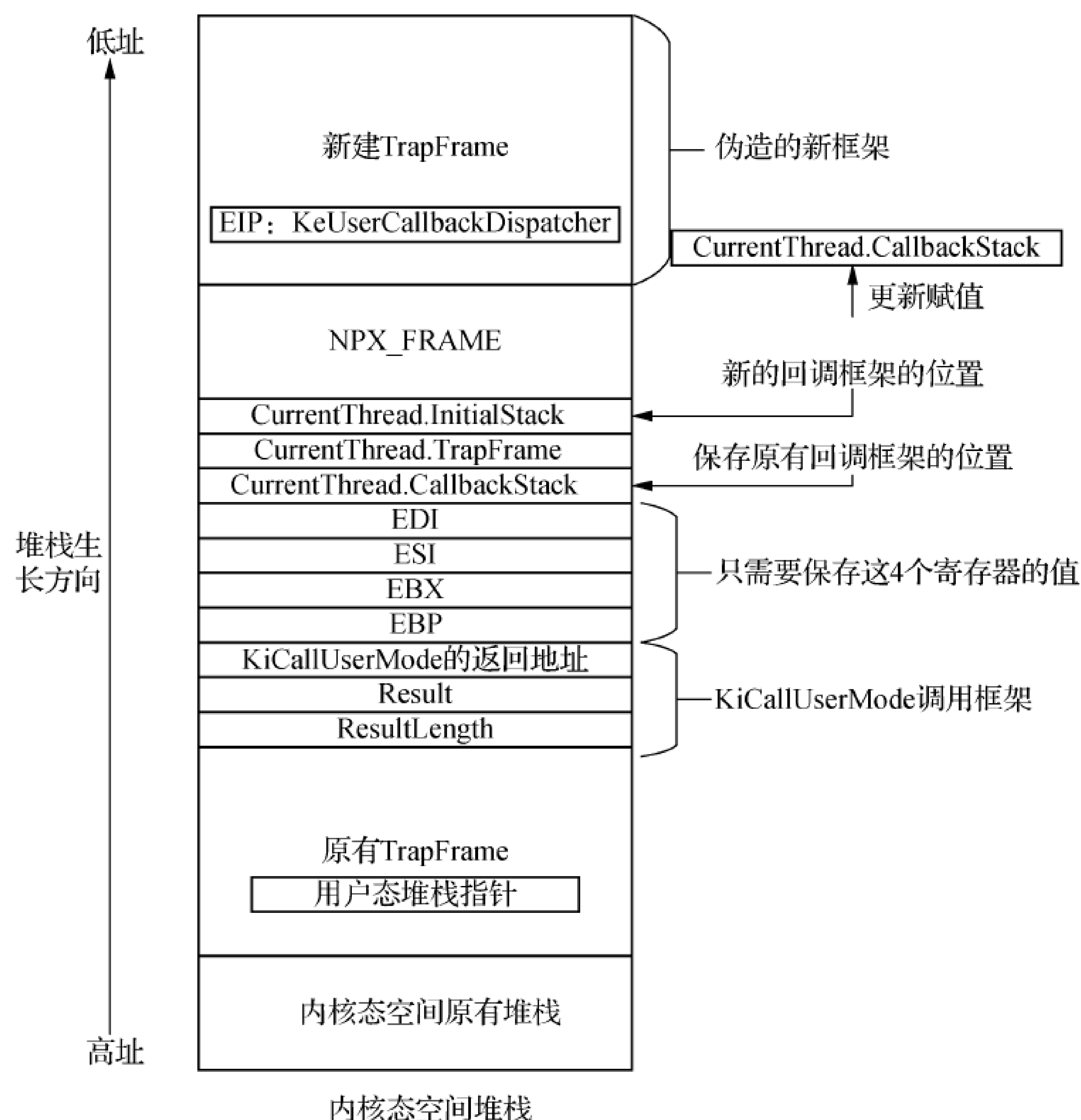


图 8－14　KiCallUserMode 对内核态空间堆栈的改造

这里要注意的是，内核堆栈的初始空间以及可写页面数量可能没有那么大，这时候需要先扩展堆栈，大约为 3 个内存页面（12 KB）的大小，以防堆栈数据溢出。

内核态堆栈构筑完成后再调用 KiServiceExit 跳转到用户态空间，并执行内核态堆栈的 TRAP_FRAME 中的指令指针 EIP，即 KeUserCallbackDispatcher 函数，该函数以用户态空间堆栈中事先构筑好的 Index、Argument 指针、ArgumentLength 等为参数，其执行过程如图 8－15 所示。

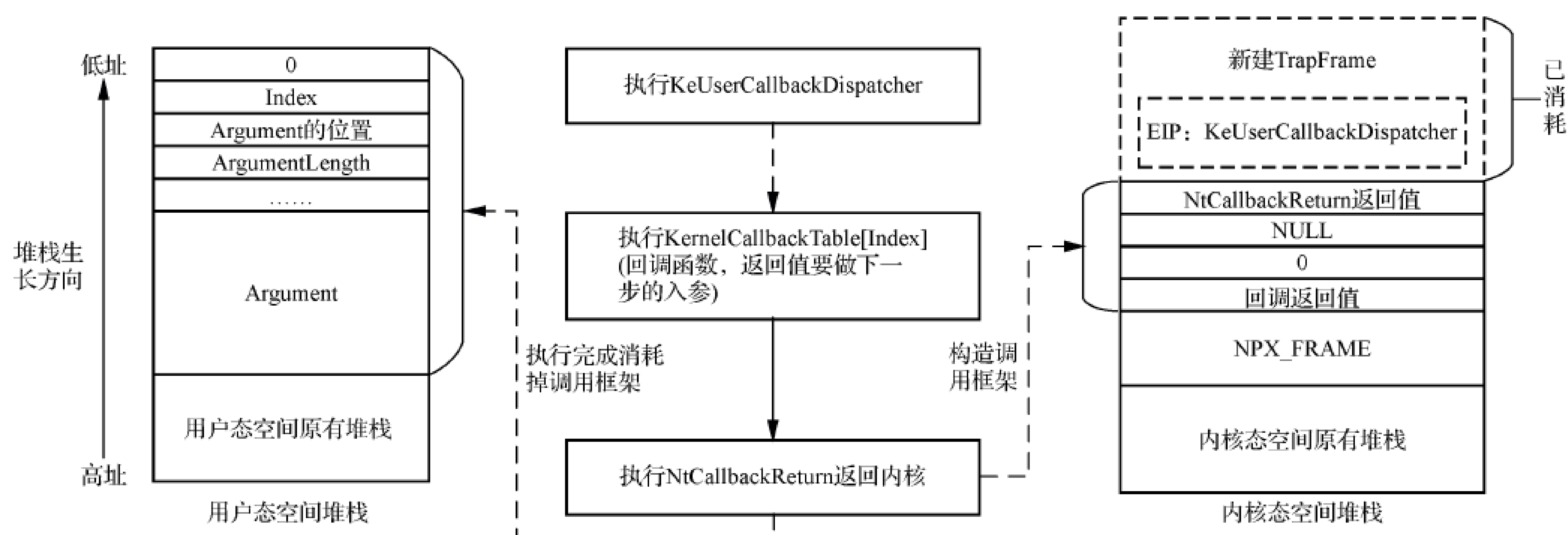


图 8－15　KeUserCallbackDispatcher 执行过程以及堆栈使用情况

KeUserCallbackDispatcher 调用返回函数 ZwCallbackReturn。但在调用 ZwCallbackReturn 之前，首先需要调用 PEB 中的 KernelCallbackTable 域对应的 Index 的元素（回调函数指针），这个调用以 Argument 指针和 ArgumentLength 为参数，调用完成后以其返回值为参数再调用 ZwCallbackReturn。

ZwCallbackReturn（NtCallbackReturn）调用是个系统调用，该调用是要返回内核态的，因为 KernelCallbackTable 回调函数的执行是在用户态空间，用的也是用户态空间堆栈，执行完成后，用户态空间堆栈配平，参数被消耗掉并出栈，用户态空间堆栈又恢复到 KeUserModeCallback 执行之前的状态了。

NtCallbackReturn 执行时，原先新建的 TrapFrame 已经消耗完了，由于接下来要返回内核态空间执行，因此要在内核态空间堆栈构筑 NtCallbackReturn 的调用框架（如图 8－15 中右半部分）。NtCallbackReturn 的主要任务有：

- 恢复当前线程的 KTHREAD 结构的 InitialStack 域为原来的值。
- 恢复 NPX_FRAME 框架中浮点寄存器中的值。
- 弹出内核态堆栈中的 NPX_FRAME，恢复到最原始的内核态空间堆栈状态，并将栈顶指针赋值给 ESP 寄存器。这里要注意的是，KiCallUserMode 是不返回的，因此它在内核态空间的回调框架不会自己清理，只能依靠 NtCallbackReturn 来清理了。“一只羊是赶，两只羊也是放”，内核态空间堆栈的恢复就干脆由 NtCallbackReturn 一并做了。
- 恢复 TSS 的 ESP0 指针（内核态堆栈为了返回用户态空间，构筑了新的 NPX_FRAME 和 TRAP_FRAME，因此堆栈指针发生了变化，此时需要将新的堆栈栈顶地址赋值给 TSS 的 ESP0 域，因为在这种状态下当前线程可能被中断和切换，既然可能发生切换，那么内核态堆栈栈顶是一定要保存到 TSS 中的）。

在上述过程中，我们用的索引值可能是 USER32_CALLBACK_WINDOWPROC，那么对应的函数就是 User32CallWindowProcFromKernel。当然，User32CallWindowProcFromKernel 也是个统一入口的函数，其本质还是调用 IntCallWindowProcW 进而调用用户态空间的视窗函数。如索引值是 USER32_CALLBACK_HOOKPROC，则对应的函数就是 User32CallHookProcFromKernel 了。从这些函数名称的后半部分（FromKernel）也可以看出这些函数是需要从内核态空间返回用户态空间去执行的。

上述过程执行完毕，向用户态空间的回调便完成了。

最后要说明的是，视窗型报文的传递过程可能会有“挂钩点”存在。挂钩（Hook）是一种特殊的消息处理机制，它可以监视系统或者进程中的各种事件消息，截获发往目标视窗的消息并进行处理。“挂钩”一词形象地描述了这种机制，就像是一把钩子钩在了流转消息的链路上，从而使消息流转的方向做了转变。当然，必须是视窗线程才会有能用于挂钩的视窗型报文挂钩表。

Windows API 提供了 SetWindowsHook 和 SetWindowsHookEx 两个方法在消息传递路径上安装挂钩。不过，只有固定的消息才能安装挂钩，例如 WH_KEYBOARD 或 WH_MOUSE，表

示在发生了 WH_KEYBOARD 消息的时候采用另一个函数来处理这个消息,其优先级比原来注册的视窗函数高。

SetWindowsHook 或 SetWindowsHookEx 实质上也是调用 win32k. sys 的扩展系统调用 NtUserSetWindowsHookEx。在 win32k. sys 中有个挂钩表,其中存放了若干 HOOK 结构,一个挂钩对应一个 HOOK 结构。并且挂钩表中对于每一个能挂钩的点都有个挂钩队列,队列中存放 HOOK 结构。也就是说挂钩表是一个队列数组,这个数组的索引就是 WH_KEYBOARD 这样的消息报文索引值,因此每个队列代表了一个消息挂钩点。

HOOK 结构中包含了所属线程、挂钩处理函数指针等参数。之所以每个挂钩点对应一个队列是因为每个点可以安装多个挂钩,最后安装的挂钩会最先被调用。挂钩表分为两种,即本线程的局部挂钩表和系统的全局挂钩表,本线程的报文队列 USER_MESSAGE_QUEUE 结构中有个挂钩表指针,指向本线程的局部挂钩表;全局指针 GlobalHooks 则指向全局挂钩表。

前文说过,在 Co_MsqDispatchOneSentMessage 函数中,一个报文在流转时遇到一个挂钩点会调用 Co_Hook_CallHooks 或 Co_IntSendMessage,而且 Co_Hook_CallHooks 在前,这也意味着 Co_Hook_CallHooks 会先被执行,后面还调不调用 Co_IntSendMessage 就要看 Co_Hook_CallHooks 的处理逻辑了。Co_Hook_CallHooks 首先判断该点位是否有局部挂钩表,没有的话再看看全局挂钩表,这也可以看出局部挂钩的优先级高于全局挂钩。然后判断这个挂钩是否就属于当前线程,如果不属于当前线程而且挂钩点比较底层,那本线程处理不了,就通过 Co_MsqSendMessage 发往 HOOK 结构所属线程,让它们去处理;否则,就在当前线程处理了,即由 Co_IntCallHookProc 完成用户态空间的挂钩函数调用。不出所料, Co_IntCallHookProc 也是调用 KeUserModeCallback,只不过入参的 Index 变成了 USER32_CALLBACK_HOOKPROC 了,但调用过程机制与上文所述是完全一致的。

8.4 键盘与鼠标报文的响应机制

键盘与鼠标消息报文的投递是视窗型报文的接收与发送的最典型示例,只不过它们都是硬件报文或张贴型报文。我们在本节中考察一下这两种设备的报文产生和投递,作为对前面内容的补充。

在 win32k. sys 模块初始化的时候,系统为键盘和鼠标各设置了一个内核线程,如下所示:

- **键盘报文线程**:KeyboardThreadMain;
- **鼠标报文线程**:MouseThreadMain。

这两个线程面向系统中其他所有线程提供键盘和鼠标的输入报文。我们先来看键盘报文线程 KeyboardThreadMain,如图 8-16 所示,其处理流程非常简单。

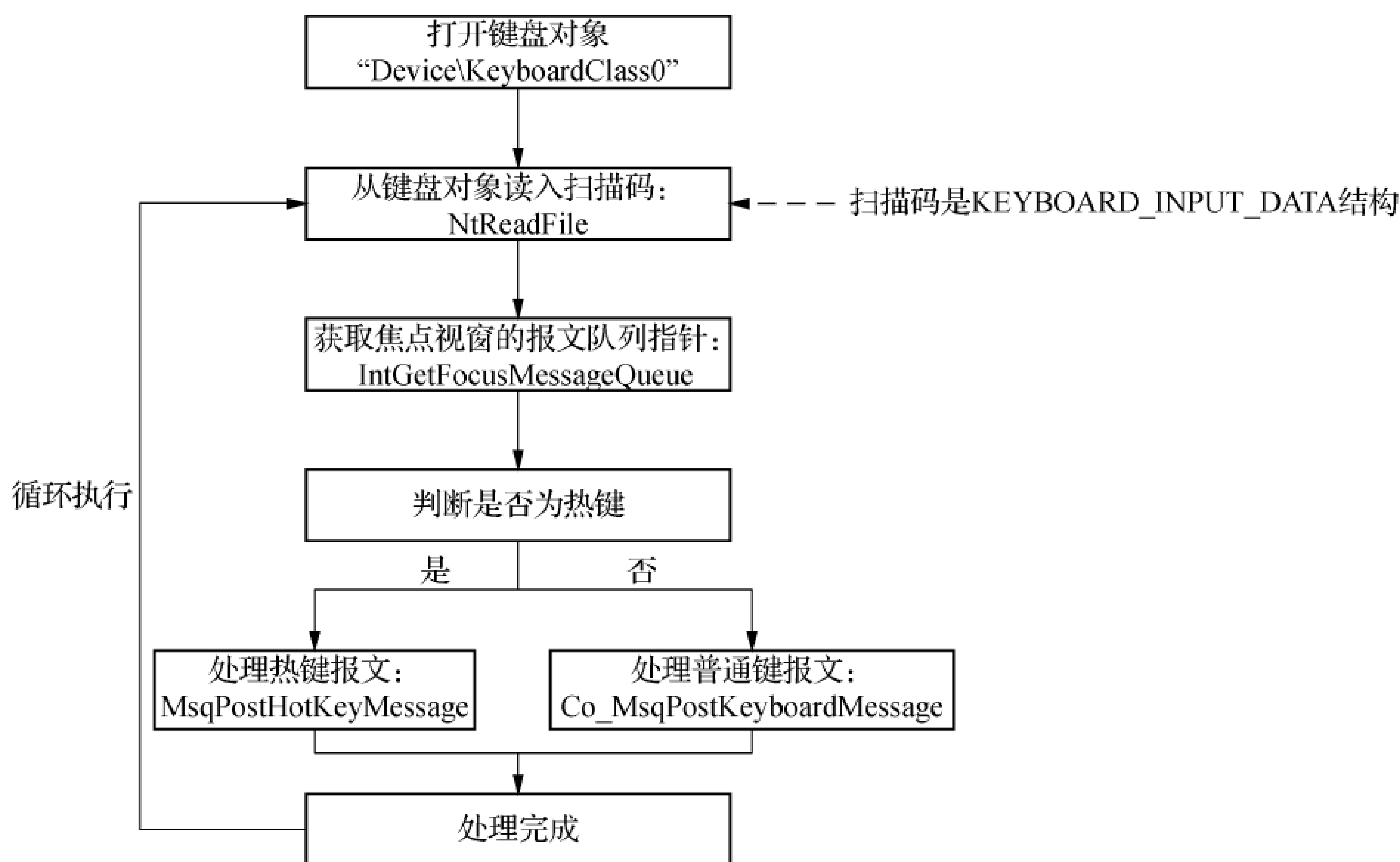


图 8－16　键盘报文线程的处理流程

这里需要说明以下几点：

- win32k. sys 模块中有个全局指针 InputDesktop 指向当前桌面的对象结构 DESKTOP_OBJECT，该对象包含一个 ActiveMessageQueue 指针指向当前的焦点视窗的报文队列（USER_MESSAGE_QUEUE 结构），系统调用 IntGetFocusMessageQueue 就是为了获取这个指针。
- 所谓热键就是一些功能组合键，例如 Ctrl + Alt + Del 组成的热键可以调取任务管理器。热键是需要提前注册的，系统中的默认热键组合是系统初始化时注册的，使用 user32. dll 提供的接口 RegisterHotKey 可以向 win32k. sys 注册热键。win32k. sys 模块中有个全局的热键列表 gHotkeyList，每次注册热键就挂入一个热键数据结构 HOTKEYITEM。
- 普通的键盘报文有 4 种类型：WM_KEYUP、WM_KEYDOWN、WM_SYSKEYUP 和 WM_SYSKEYDOWN，代表了键盘的按压和弹出。这些报文都是采用 MsqPostMessage 函数投递出去的。当然，投递之前要先检查是否有挂钩，有的话就先执行挂钩函数。

下面再来看鼠标报文线程 MouseThreadMain，如图 8－17 所示，其处理流程与键盘报文线程的处理流程很不一样。

前面的流程比较简单，我们从 SendMouseEvent 开始解释。SendMouseEvent 的参数是个 MOUSEEVENT 结构，这个数据结构中包含了鼠标移动的坐标 X，Y，如果 X，Y 值与上一次的 XY 值不一样，则说明鼠标发生了移动。SendMouseEvent 包含两个操作：执行 IntMouseInput 和 ClearMouseInput，前者是鼠标移动时发送报文的操作载体，后者只是将 MOUSEEVENT 结构清零。

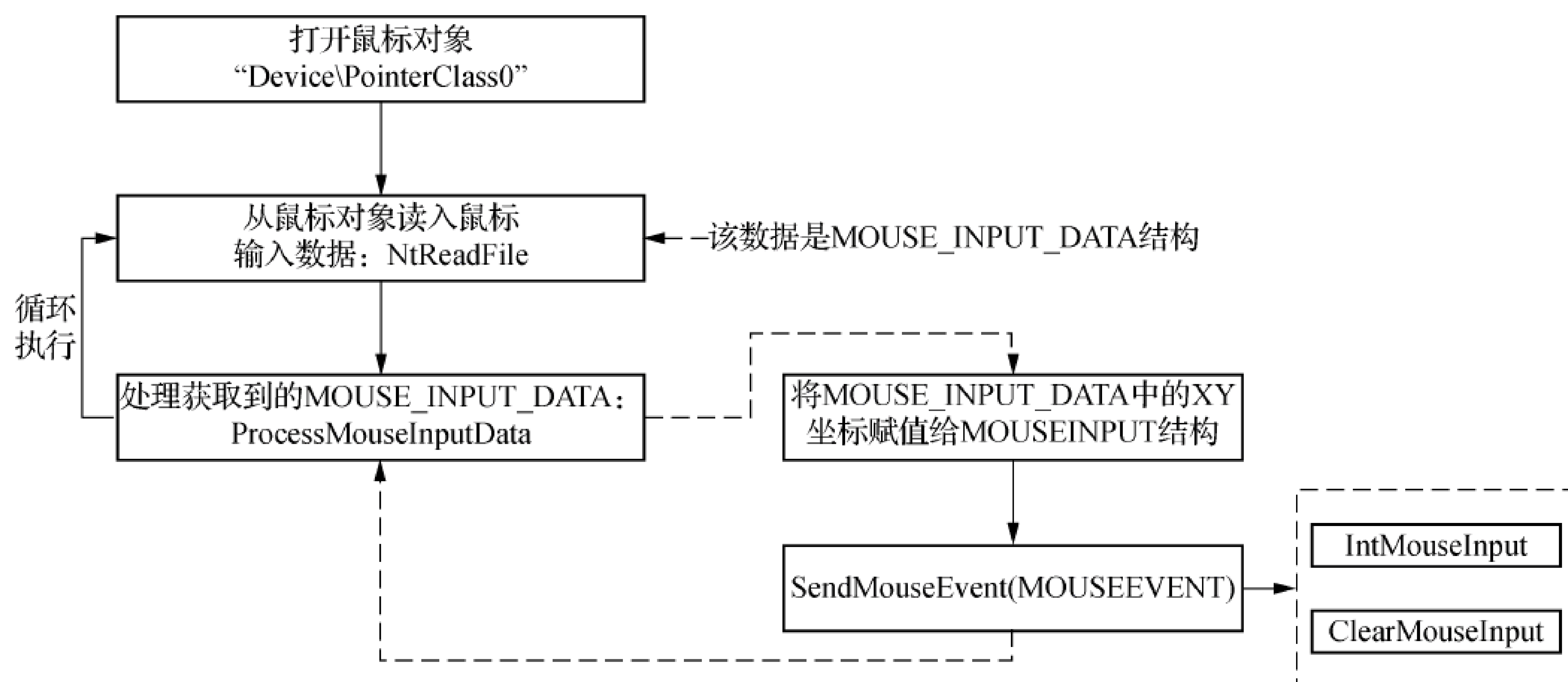


图 8－17　鼠标报文线程的处理过程

IntMouseInput 包含以下三个操作：

- 获取当前桌面上光标的位置：IntGetCursorLocation。
- 判断光标是否发生了移动，判断依据是将当前 MOUSEEVENT 的 X，Y 坐标与上一次的 X，Y 坐标相比较，若有不同则说明发生了移动，调用 IntEngMovePointer 移动鼠标；反之，则说明没有移动，不需要进行任何操作。
- 若发生了移动则通过 MsqInsertSystemMessage 将一个鼠标移动报文插入系统报文队列 SystemMessageQueue 中（如果队列已满就不再插入），并唤醒在硬件消息事件 HardwareMessageEvent 上等待的线程。

鼠标报文有这样几种类型：WM_MOUSEMOVE、WM_LBUTTONUP、WM_RBUTTONUP、WM_LBUTTONDOWN、WM_RBUTTONDOWN，分别表示左、右键的按下和弹起以及滚轮移动，这几类报文均存放在 SystemMessageQueue 中。SystemMessageQueue 是个环形队列，其最大值 SYSTEM_MESSAGE_QUEUE_SIZE 在 Windows NT 内核中定义为 256。

我们在前文中介绍过，Co_MsqFindMessage 在处理完张贴型报文后开始处理硬件报文，这里说的就是鼠标报文。对于来自鼠标的报文，我们用 Co_MsqTranslateMouseMessage 来投递，这种投递就是将报文挂到目标视窗线程的 HardwareMessagesListHead（硬件报文队列）中，并唤醒其报文队列 MessageQueue 的 NewMessage 事件，表明有新的消息到来。当然，在调用 Co_MsqTranslateMouseMessage 投递报文前也要先看看挂钩点是否有挂钩项，有的话要先执行挂钩函数，能不能轮到 Co_MsqTranslateMouseMessage 投递就要看挂钩函数对于报文的拦截操作处理了。

这里还要强调一下：SystemMessageQueue 环形队列是由内核线程 MouseThreadMain 来填充的，作为内核线程，MouseThreadMain 是运行在任意上下文的，它与具体视窗的硬件报文队列要有个交接，这个交接由全局硬件报文队列 HardwareMessageQueueHead 完成，只有没有被挂钩函数“勾走”的报文才会有幸进入这个全局硬件队列，进而被投递给具体视窗线程。

以上就是键盘与鼠标报文的产生和投递的处理过程。

本章小结

本章首先介绍了视窗型报文的各种数据结构和 7 种报文队列,然后介绍了视窗型报文的接收和发送流程,也就是报文在各种队列中是怎样流转的。当报文返回用户态空间时,需要借用回调机制进行处理,同时在返回的过程中,需要处理好用户态空间和内核态空间堆栈的布局。最后以键盘和鼠标的中断响应为例简单回顾了视窗型报文的处理流程。

第 9 章　结构化异常处理

结构化异常处理(Structure Exception Handling,SEH)是 Windows 内核机制的重要组成部分。SEH 本身与语言无关,但现代编程语言(如 C++ 、C#、Java 等)的异常处理框架都是在 SEH 基础上封装和演变过来的。因此,了解结构化异常处理的框架及相关流程,对于掌握 Windows 和编程语言的异常处理机制非常重要。

但是我们要明白,所谓“异常”是相对而言的。什么是异常? 内存缺页也算是异常,但这是系统中经常会发生的事件,这类事件就不能算异常,而是常态。此处所说的异常主要是指诸如除数为 0、内存违例读写、内存访问溢出此类的行为,针对这类异常我们希望能够妥善且优雅地解决问题,比如跳转到正常的代码区或者修复执行参数等,因此需要用到结构化异常处理机制。但是异常处理是个双刃剑。首先,针对某些异常确实可以通过修改参数来妥善处理,使之“拨乱反正”,回到正常的执行流程中,但是针对大多数的异常,其主要作用还是优雅地终止出错线程或进程;其次,由于异常处理框架涉及复杂的堆栈操作和善后处理,因此执行开销非常大、耗时长、复杂度高,不能大规模频繁使用。

除了结构化异常处理,Windows XP 及以上的版本还提供了一种叫作向量化异常处理(Vectored Exception Handling,VEH)的机制。VEH 机制是对 SEH 机制的补充完善,提供了优先于 SEH 的处理机会。

在本章我们将按照图 9 - 1 所示的提纲进行结构化异常处理机制的介绍。

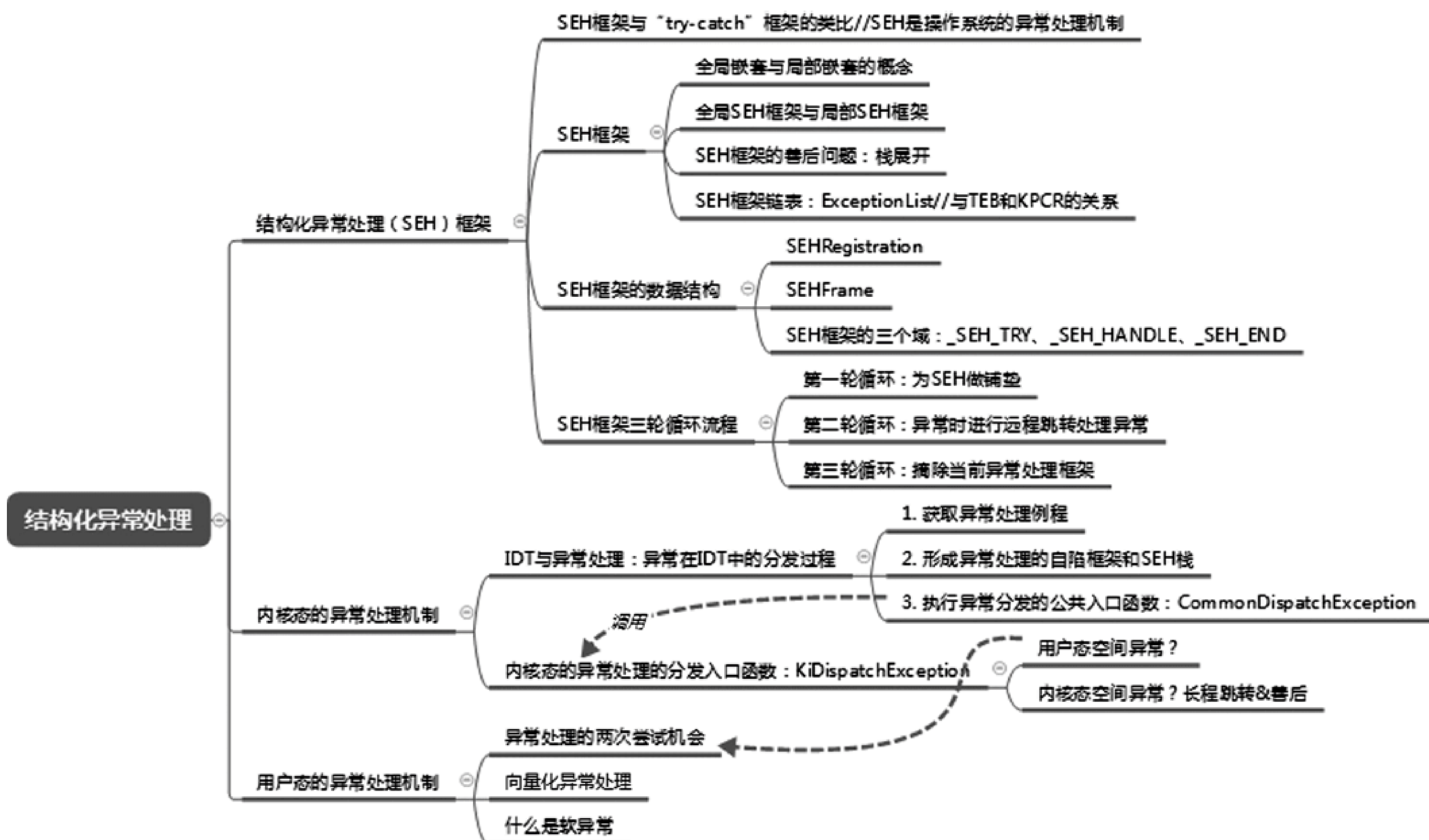


图 9 - 1　本章提纲

9.1　结构化异常处理框架

我们在使用 C++ 等语言编程时经常遇到“try-catch”框架，这个框架类似以下形态：

```
try {
    Func1();
    } catch(…){
    Func2();
 }
```

其实这是高级语言对结构化异常处理机制的改进和封装，究其本质还是操作系统的 SEH 框架，而 Windows 的 SEH 框架是下面这个样子：

```
_SEH_TRY{
    Func1();
}_SEH_HANDLE{
    Func2();
}_SEH_END
```

可以看到，这个框架与高级语言的“try-catch”框架长得很像，“本是同根生，变异何太急”，理解了 SEH 框架，对于高级语言的异常处理机制也就大概清楚了：

- **_SEH_TRY** 后面近邻的大括号里的内容是我们要保护的程序代码，这个大括号称为“保护域”。说是保护，其实不是说不让发生异常，而是异常发生后有应对的处理机制，因此我们要把可能发生异常的代码放在_SEH_TRY 域内，它对应“try-catch”框架的 try 域。
- **_SEH_HANDLE** 后面邻近的大括号里的内容是异常代码的“处理域”，这个域内的代码负责对异常进行过滤处理。异常的种类是很多的，例如除零异常、内存访问异常等，不可能有一个大而全的处理函数把所有的异常处理都囊括在内，因此需要根据异常的种类分类指导、精准处理，构筑一个个小而美的异常处理函数。_SEH_HANDLE 域的工作就是根据异常的种类找到对应的处理函数，它对应的是“try-catch”域的 catch 域。
- **_SEH_END** 是个宏定义，负责的是善后工作。如果保护域中的代码未发生异常，正常执行完毕，_SEH_END 就从局部 SEH 框架栈中摘除当前 SEH 框架，并从全局 SEH 框架队列中摘除相应框架。

所以 SEH 框架就是先执行_SEH_TRY 域中的代码(保护域)，执行过程中如果发生异常则转去执行_SEH_HANDLE 域中的代码(处理域)，否则就跳过_SEH_HANDLE 域直接执行_SEH_END 域中的代码来摘除 SEH 框架(善后域)。

那么什么是 SEH 框架？什么是全局 SEH 框架？什么又是局部 SEH 框架呢？

一个 SEH 框架对应一个_SEH_TRY 和_SEH_HANDLE 域，对应到高级语言就是一个 try 域和对应的若干个 catch 域。在实际应用中，我们会遇到嵌套的情况，例如 try 域中还有个 try 域，或者 try 域中的函数里面又调用了“try-catch”框架，这种叫作异常处理嵌套。

嵌套分为两种：局部嵌套和全局嵌套。所谓局部嵌套，是指能够一目了然的，即上面描述的 try 域中还有个 try 域的情况，这种嵌套也叫“形式嵌套”；而 try 域中的函数里面又调用了“try-catch”框架的情况称为全局嵌套，也叫“实质嵌套”，实质嵌套并不能一目了然。局部嵌套对应的框架叫作局部 SEH 框架，全局嵌套对应的框架叫作全局 SEH 框架。

由于异常处理存在嵌套的情况，并且异常处理要用到堆栈结构，因此也会形成 SEH 框架栈，与嵌套种类相对应地也分为局部 SEH 框架栈和全局 SEH 框架栈。在栈中，越在内层的框架越靠近栈顶，也就越早会弹出。

图 9－2 中左半部分是 SEH 框架栈在堆栈中的示意图。假定包含外、中、内三层 SEH 框架，当异常发生时，先从内层 SEH 框架开始过滤处理，内层处理不了时由中层 SEH 框架来处理，当中层也处理不了时再调用外层 SEH 框架来处理，逐层向外扩展。当由中层或外层 SEH 框架处理时，跨越过去的这几层 SEH 框架（内层、中层）存在一个“栈展开”的善后问题。

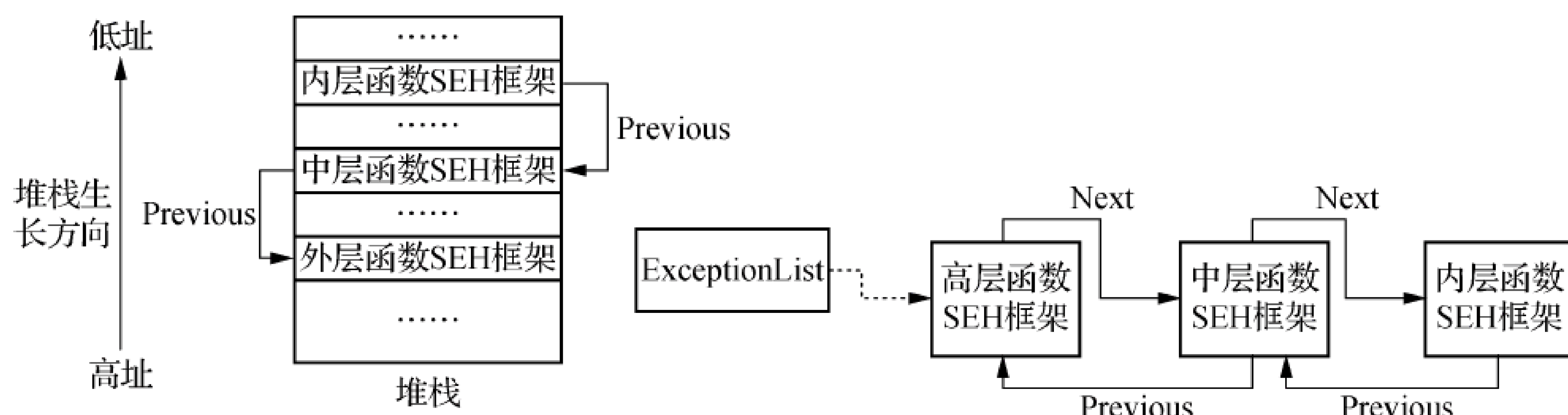


图 9－2　全局 SEH 框架在堆栈和异常链表中的形态

因为在代码从外到内执行的过程中，SEH 框架也是由外而内被逐层设置的，SEH 框架的设置就像盖楼时搭建脚手架一样是用来预防发生异常或意外的，但搭建脚手架是需要成本的，会有资源的占用和消耗，而现在越过了内层 SEH 框架就相当于上层的脚手架作废了，那么之前占用的资源就应该被释放。正常情况下异常的过滤和处理之后会有对于资源的释放，但现在越过了一部分过滤和处理阶段，资源释放这些善后工作自然也会被越过，这就是“栈展开”问题的本质。

图 9－2 中右半部分是 SEH 框架在异常框架列表 ExceptionList 中的形态，内层 SEH 框架最后入队，因为内层的代码是最后执行的。对 ExceptionList 大家可能还有印象，我们在保存线程现场的时候会压栈当前线程的执行上下文，其中就包括这个 ExceptionList。在用户态空间，这个链表是跟线程相关的，由 TEB 的线程信息块描述；在内核态空间，这个链表是由 KPCR 的线程信息块描述的，是全局的。但无论怎样，在用户态空间和内核态空间都要有这个队列，而且都是由 FS 寄存器的相同偏移（Offset：0）位置指向的，这就为我们在两个内存空间中获取异常处理链表提供了便利，如图 9－3 所示。

这里要强调的是，只有全局 SEH 框架才会入队成为 ExceptionList 中的节点（横向生长），也只有全局 SEH 框架才会进入堆栈形成链表（纵向生长）。形式的、局部的 SEH 框架除了最外一层作为全局 SEH 框架入队、入栈之外，其内层的嵌套 SEH 框架是作为队列元素

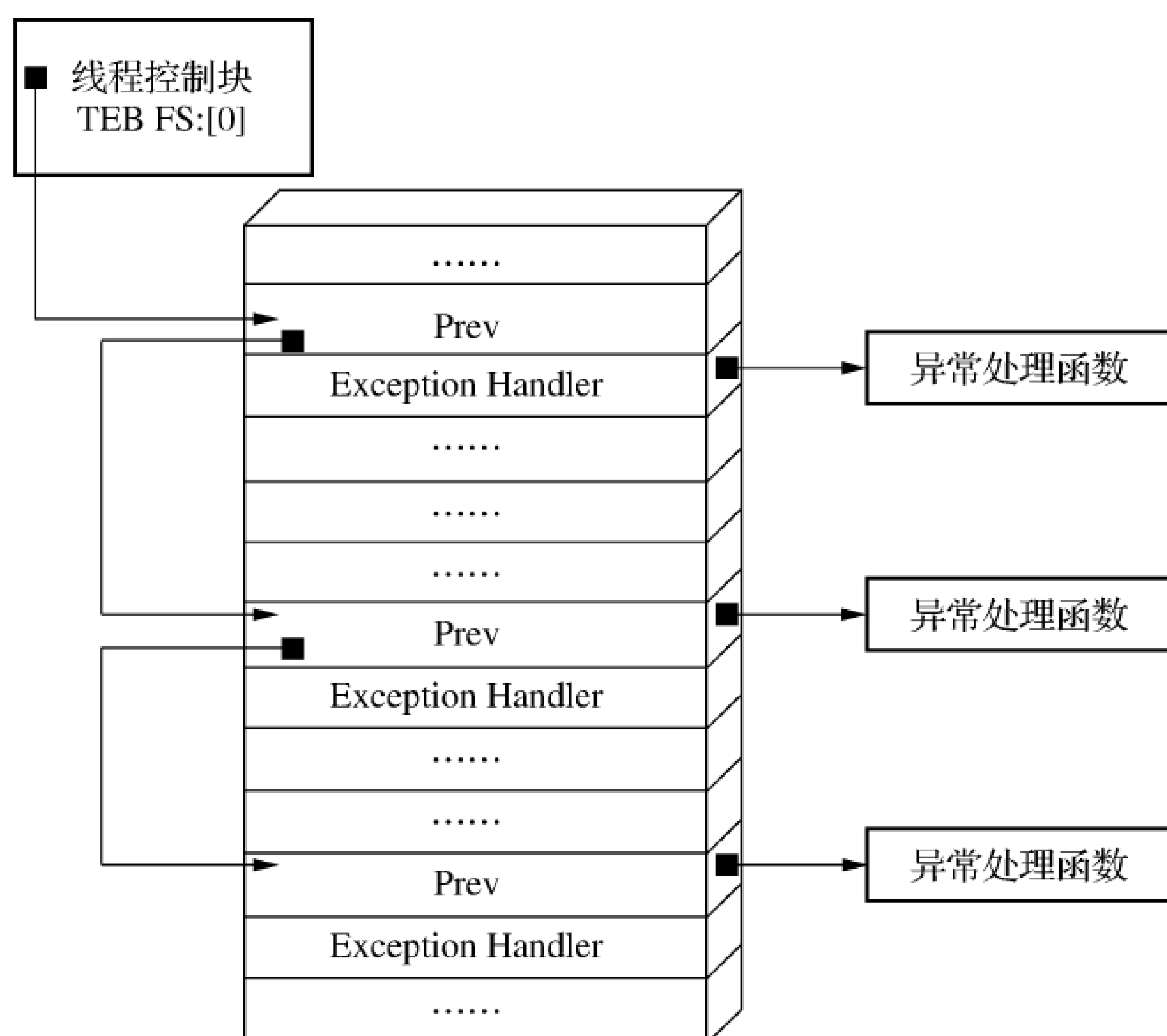


图 9－3　TEB 指向异常处理链表

的兄弟节点存在的(纵向的),但本质上也构成一种堆栈结构,因此也叫"框架栈",只不过一个是全局的,一个是局部的,范围不一样,如图 9－4 所示。

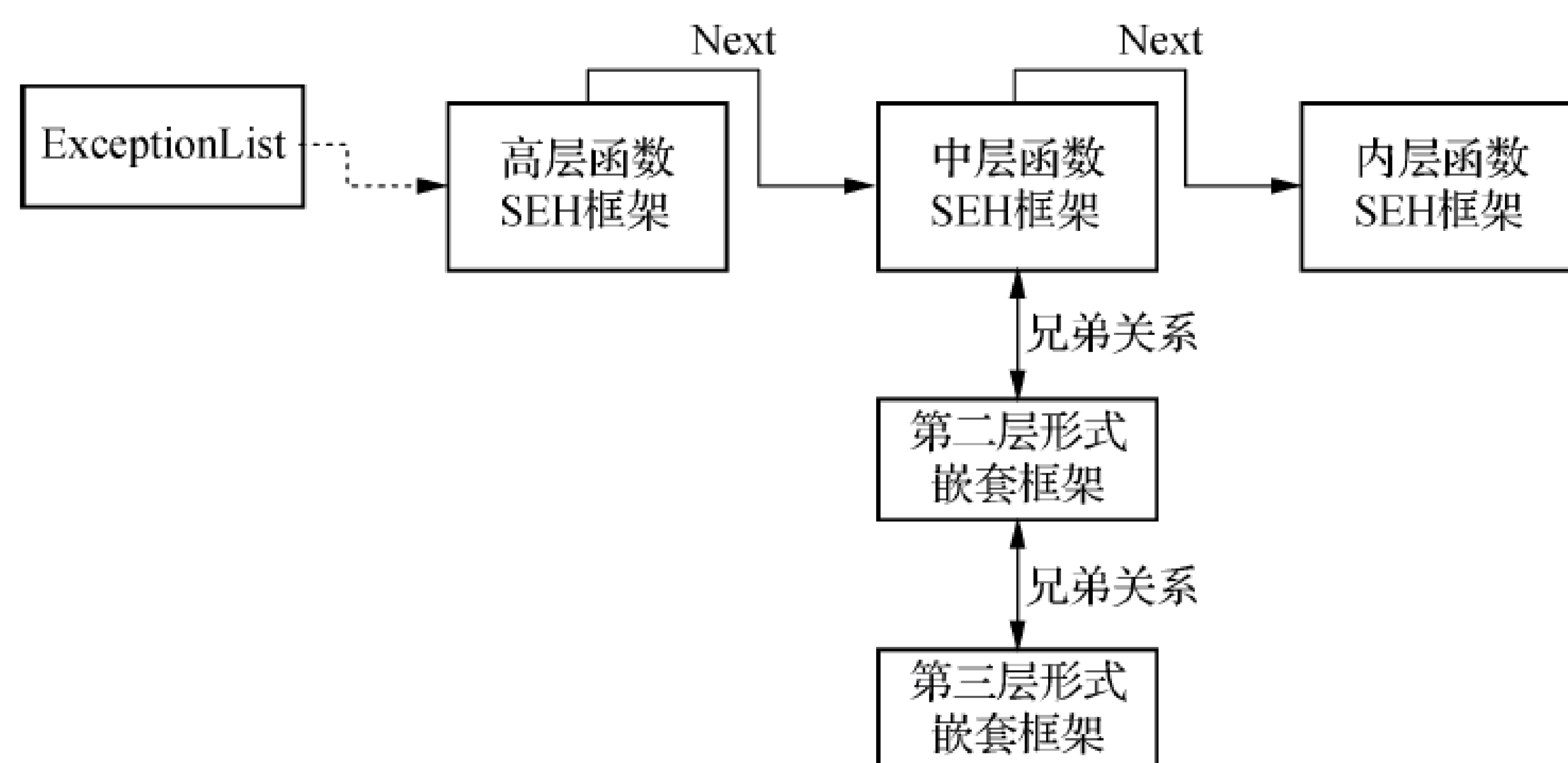


图 9－4　局部 SEH 框架在异常链表中的形态

系统通过_SEHEnterFrame 方法将一个全局 SEH 框架挂入 ExceptionList,通过_SEHUnregisterFrame 方法从 ExceptionList 中摘除全局 SEH 框架。而挂入 ExceptionList 的是一个_SEHRegistration 数据结构,它包含两个元素:

- 低址端的是_SEHRegistration 指针,指向上一个_SEHRegistration 结构(该结构代表相当于当前框架更外层的 SEH 框架)以形成链表;
- 高址端的是 SEHFrameHandler 函数指针,指向异常处理实施函数,默认是 SEHCompilerSpecificHandler,用于长程跳转等操作,而长程跳转的目的地址就是 SEH _HANDLE 域。

在_SEHRegistration 的外围还扩展了其他的数据结构，包括了过滤函数指针、善后函数指针等，以研判本框架是否认领该异常以及怎样善后资源等。SEHCompilerSpecificHandler 执行时会首先判断这个框架是单个全局 SEH 框架还是同时包含局部 SEH 框架：

（1）如果是单个全局 SEH 框架，则：

- 调用框架的过滤函数以确认是否认领本次异常；
- 如果认领则首先调用所有内层 SEH 域的善后函数，否则继续向上回溯直至找到认领本次异常的 SEH 框架，步骤同前；
- 如果认领则执行本 SEH 域的长程跳转。

（2）如果同时包含局部 SEH 框架，则：

- 从后向前调用当前全局 SEH 框架的局部框架栈，执行过滤函数以确认是否认领本次异常；
- 如果认领则首先调用所有已跨越的内层 SEH 域的善后函数，否则继续向上回溯直至找到认领本次异常的 SEH 框架；
- 如果当前全局节点的局部 SEH 框架栈遍历完也没有找到能处理当前异常的局部 SEH 框架，则在全局 SEH 框架队列中向上回溯继续寻找，步骤同前；
- 若找到则执行本 SEH 域的长程跳转。

同时在栈中也是这样一种结构，也会通过_SEHRegistration 指针形成链式结构。可以想象，_SEHRegistration 结构代表了一个全局 SEH 框架节点，其前一个节点代表更外一层的全局 SEH 框架。而对于局部嵌套，_SEHFrame（_SEHRegistration 代表更外层的框架，_SEHRegistration 包含_SEHFrame）则代表了一个局部 SEH 框架，_SEHFrame 内部有个队列，存放了嵌套的局部 SEH 框架（本质上是 SCOPETABLE 数组，SCOPETABLE 结构代表局部 SEH 框架）。这还只是其中的两个数据结构，由于涉及 SEH 机制的数据结构多且烦琐，我们在这里不详细介绍这些数据结构，只是在用到的时候再提及相关内容。

SEH 框架的跳过是通过_SEH_HANDLE 机制实现的。_SEH_HANDLE 实质上是个宏定义，具体值是_SEH_EXCEPT（_SEH_STATIC_FILTER（返回值））。_SEH_STATIC_FILTER（返回值）作为_SEH_EXCEPT 的参数，可以由系统来定义，也可以由应用程序自己定义。Windows 定义了三个值：

- **_SEH_CONTINUE_EXECUTION**：定义为 -1，表示忽略本次异常；或本次异常已经解决，程序流继续执行。
- **_SEH_CONTINUE_SEARCH**：定义为 0，表示本框架不认领本次异常，继续向上回溯到上一层 SEH 框架。
- **_SEH_EXECUTE_HANDLER**：定义为 1，表示认领本次异常，并实施长程跳转，即执行_SEH_EXCEPT 域中的代码，也是_SEH_HANDLE 后面域中的代码。

由此可以看出_SEH_HANDLE 起着过滤和处理异常的作用。这里要强调一点，不是所有的异常都会先进入_SEH_HANDLE 域。因为异常发生后，首先会由操作系统内核的异常响应函数进行异常分配，对于缺页等假异常会由底层的处理函数处理，而且如果进程处于被

调试状态,异常发生后也会由调试器先接管异常,根本轮不到上层的SEH框架来处理,只有底层和调试器都不管的异常才会由SEH框架来接管。这些情况是比较常见的,例如我们在使用VC 6调试代码的时候,一旦发生异常会先在VC 6的调试窗口中中断,并输出当前寄存器等上下文,根本轮不到用户自己定义的异常处理函数来接管。

综上所述,SEH是一个比较复杂的机制,除了全局和局部SEH框架,还要考虑栈展开的问题。正如前文所述,当前的SEH框架必定是栈帧中最上层的框架,或者说是队列中最深的框架,如果当前框架不能解决问题,那就要向上回溯以找到能解决问题的框架,当找到了并且也把问题解决了,那么这个SEH框架也就完成了使命,但是被它跳过去的那些SEH框架需要释放事先动态申请的资源,否则资源会溢出。另外假如代码在执行的过程中没有遇到异常,当执行到_SEH_END位置的时候,也需要执行当前SEH框架的善后函数以释放资源,并且要摘除SEH框架,这既是SEH框架的要求,也是编程良好的习惯。

SEH框架嵌套到代码中时看上去比较简单,但是被编译器编译后会生成一堆代码"附着"在保护域代码的周围,这些代码设置诸如长程跳转地址等一些初始变量,以便异常响应函数作出相应的处理。所以说SEH既是Windows基础框架,也是编译器作用的产物。

从高层视角来看,SEH编译后会形成三轮循环,如图9-5所示。

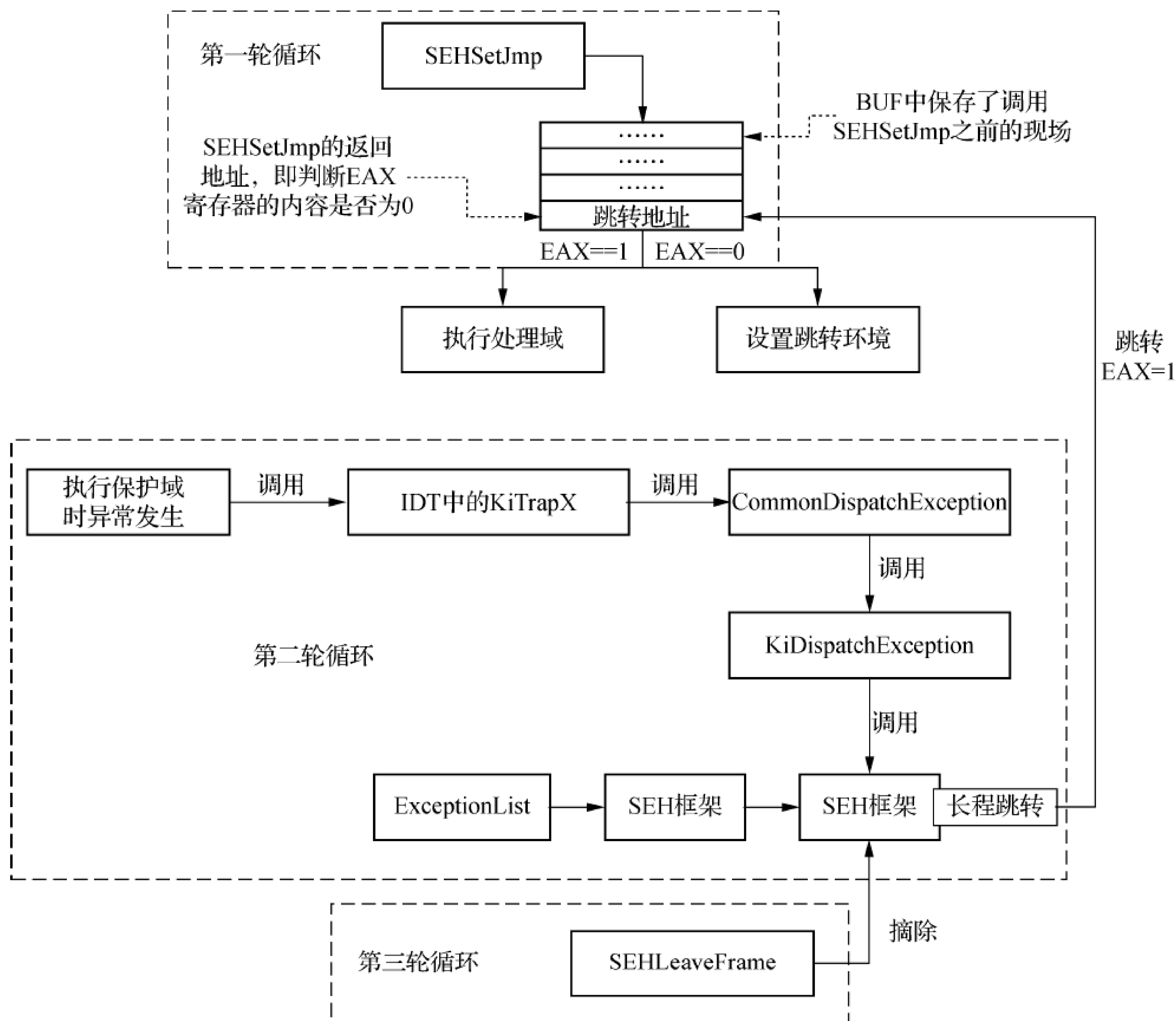


图9-5　SEH框架三轮循环流程

1）第一轮循环

在第一轮循环中，要设置两部分内容：

- 一部分是当前域的 SEH 框架及其长程跳转的统一入口（_SEHCompilerSpecificHandler）等参数，并将其挂入 ExceptionList 队列；
- 另一部分是通过 SEHSetJmp 设置长程跳转缓冲区，这个缓冲区包括长程跳转的返回地址、当前堆栈指针等，这些参数保存了调用长程跳转前夕的现场。

这里的返回地址是 SEHSetJmp 的返回地址，是一条汇编指令，即调用 SEHSetJmp 这条指令的下一条。编译器将此处的逻辑编译为一个判断，即 SEHSetJmp 的返回值（注意返回地址和返回值的不同，返回地址代表的是一条汇编指令，返回值是执行完函数后 EAX 寄存器的内容）是否为 0，为 0 就设置跳转环境，不为 0 就跳转到处理域（_SEH_HANDLE 域中的内容）。返回值为 0 就是第一轮循环要做的事情（做铺垫），不为 0 就是第二轮循环中发生异常时要做的事情（实质处理异常）。

长程跳转缓冲区保存在 SEH 框架的某个域中，而且该域必须是遍历 ExceptionList 时非常容易找到的。可以看出，第一轮循环就是为 SEH 做铺垫的。

2）第二轮循环

SEH 框架的保护域代码被编译在第二轮循环中。执行时如果没有发生异常，则万事大吉，正常结束第二轮循环并执行到_SEH_END。但如果发生了异常，则系统会通过底层的异常分发函数进行处理，中途若没有调试器的拦截，那么异常会由该类型的异常分发函数通过遍历 ExceptionList 队列找到能够处理该类异常的 SEH 结构，并通过该结构找到长程跳转缓冲区做长程跳转，来到异常处理的真正代码区（处理域），也就是第一轮循环中 SEHSetJmp 的返回地址，并且在长程跳转函数中还会将 EAX 寄存器置为 1，这意味着此时 SEHSetJmp 的返回值为 1。前文说过，SEHSetJmp 的返回值不为 0 就跳转到处理域。

当然这个代码可能只是简单地获取一下异常码或者优雅地终结出错线程，也可能是更改参数或者弹出个选择框等，但不管怎样毕竟是对异常进行处理了，第二轮循环结束。

3）第三轮循环

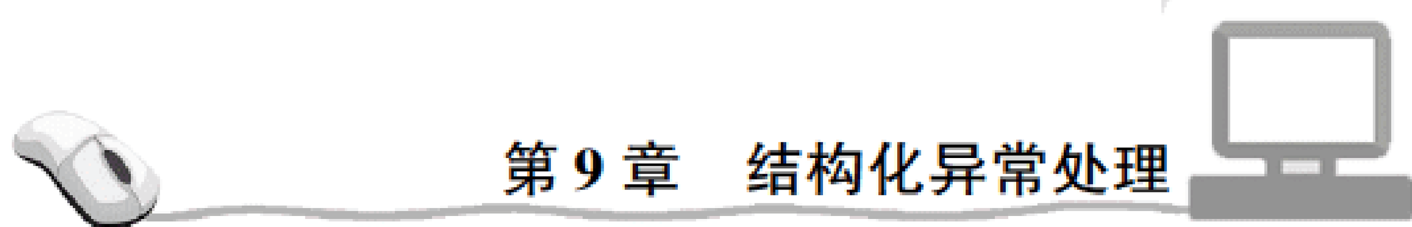
第三轮循环主要是调用_SEHLeaveFrame 从 ExceptionList 摘除当前的 SEH 框架，这就是_SEH_END 的事了。

9.2 内核态空间的结构化异常处理流程

本节我们来分析下在内核态空间的结构化异常处理流程。

在内核中，异常和中断没有什么区别，其分发机制也大致相同。就像中断有中断服务例程（ISR）一样，异常在中断描述符表（IDT）中也有相应的异常响应处理例程，如下所示：

- KiTrap0：处理除数为零异常；
- KiTrap3：处理通过断点指令 int 0x3 实现的断点异常；



- KiTrap6:处理非法指令异常;
- KiTrap14:处理内存页面异常;
- KiTrap16:处理浮点指令异常。

当然还有其他种类的异常,这里不列举那么多。发生异常的原因主要可归为三类:

- **执行指令失败引起的异常**:比如除数为 0、访问权限不足、页面映射错误等,这种异常情况发生时,push 到堆栈中的是失败的那条指令的地址,用意是异常处理完毕过后不要放弃转而再执行一遍原指令,这种异常也是我们最常遇到的。
- **int 0x3 断点异常**:这种情况发生时 push 到堆栈中的是断点指令的下一条指令的地址,这是我们在调试程序时最常遇到的。
- **严重出错异常**:这种异常是无法恢复的,例如 BSOD(Blue Screen Of Death)。

在第 3 章中我们讲过,自陷、异常、中断发生时都会在内核态堆栈中形成一个自陷框架 TRAP_FRAME,里面包含当前指令指针 EIP 等现场上下文,其实刚刚提到的 push 到堆栈中的指令地址就相当于 EIP,异常处理完毕就会执行。

IDT 中描述了异常响应处理例程,其中前 30 个都是异常处理例程,例如 KiTrap0 等。接下来我们以 KiTrap0 为例来看下异常处理流程,如图 9-6 所示。

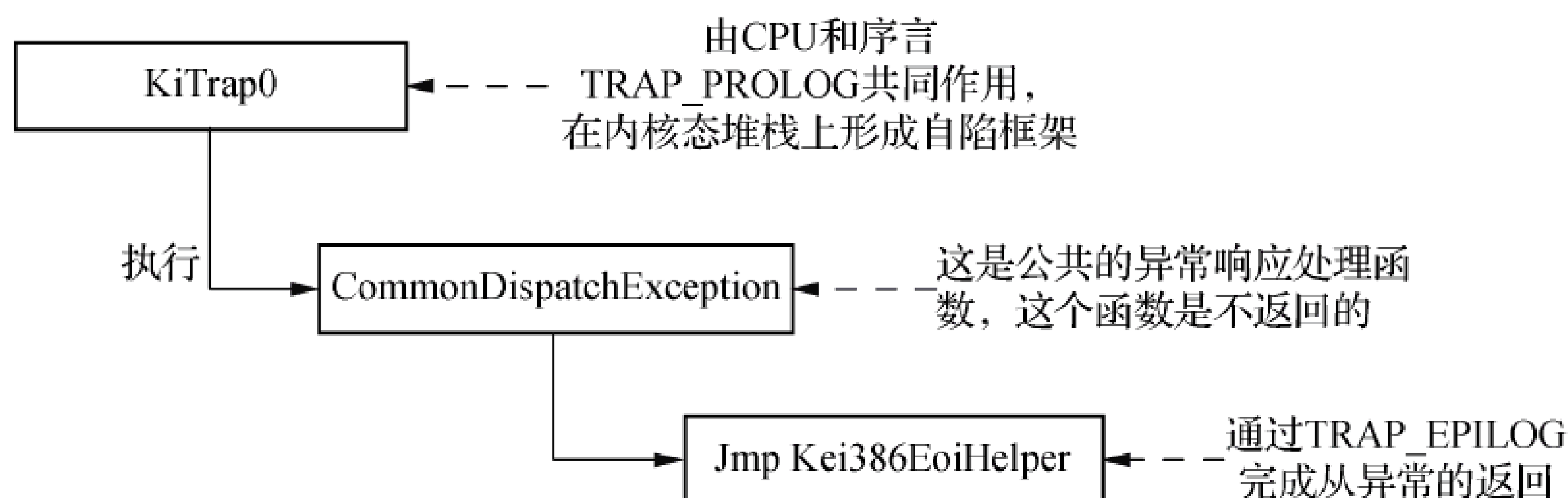


图 9-6　KiTrap0 处理异常的流程

1)获取异常处理例程

0 号异常发生时,CPU 根据异常的种类从 IDT 中获取异常处理例程(KiTrap0)。

2)形成异常处理自陷框架和 SEH 框架栈

从 KiTrap0 进来时,在 CPU 和序言 TRAP_PROLOG 的共同作用下形成了自陷框架 TRAP_FRAME,这里的自陷框架与系统调用时的自陷框架大小完全相同,结构内容上也基本一致,因为如果不一致,序言和尾声就不能通用了,并且有可能会发生堆栈无法恢复和配平的问题,这是绝对不允许的。自陷框架的结构在前文中已经有所描述,在框架中序言部分会把线程的 ExceptionList 指针压栈,但如果在异常处理的过程中又发生了异常,即发生嵌套异常,则会再生成一个自陷框架,并把当前的 ExceptionList 指针压栈,而每一个 ExceptionList 指针都指向本次异常发生时对应的 SEH 框架,这样执行下来便形成了 SEH 框架队列,或者叫 SEH 框架栈。

3)执行异常分发公共入口函数

之后执行异常分发的公共入口函数 CommonDispatchException,注意该函数是不返回的,

它用来遍历 ExceptionList 以找出能处理当前异常的 SEH 框架并进行异常处理。如果处理成功则通过一个 Jmp 跳转到 Kei386EoiHelper 函数中,这个函数通过寄存器传递参数,执行尾声操作、恢复堆栈并从异常处理返回。执行到这里意味着异常处理已经完成了,接下来需要执行自陷框架中的 EIP 指令了,EIP 指令或者是出错指令本身,或者是出错指令的下一条。

我们再以 KiTrap14 为例来讲述内存页面异常的处理流程,以便加深对 SEH 机制的理解,如图 9-7 所示。

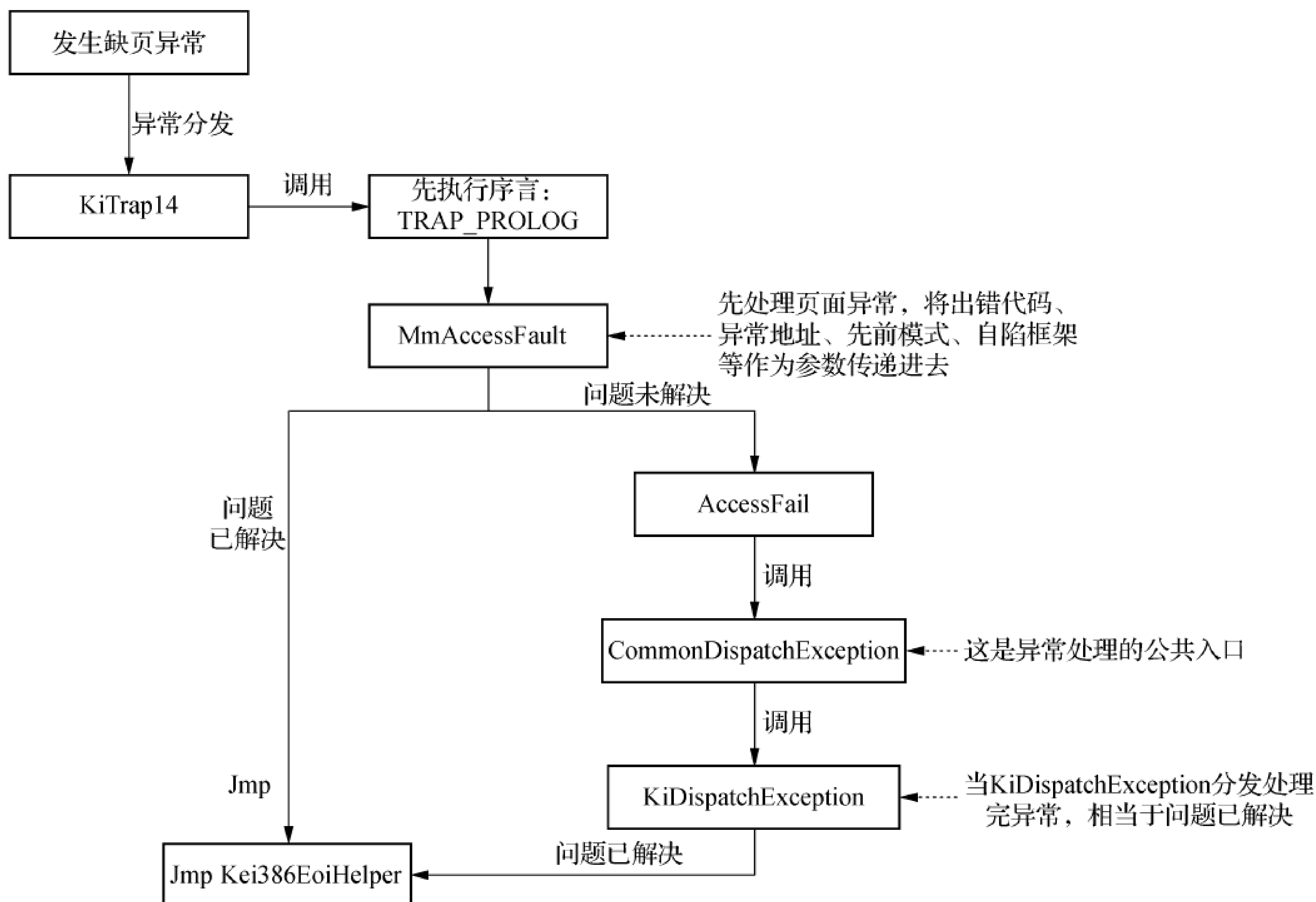


图 9-7　KiTrap14 处理异常的流程

当缺页异常发生时,进程找不到对应虚拟地址的物理内存页面,或者为缺页异常,或者为页面无映射,前者看似异常实则正常,而后者则是真正的异常。当缺页异常发生时进入异常分发例程并进入 KiTrap14,KiTrap14 首先执行序言操作 TRAP_PROLOG 以建立自陷框架等。然后调用页面异常处理函数 MmAccessFault,之所以先调用它,就是因为缺页异常大部分是正常的,是可以通过内存倒换机制解决的,这也是 Windows 内存管理的固有机制。MmAccessFault 以出错代码、异常地址、先前模式等为参数,比如异常地址的实参就是执行出错的那条指令的地址,这个地址是由 CR2 寄存器保存的,如图 9-8 所示。

- 如果在 MmAccessFault 的执行过程中异常解决了,就直接跳转到 Kei386EoiHelper,这个函数执行 APC 投递和尾声 TRAP_EPILOG、消除自陷框架并从异常处理中返回。返回后要么跳过出错地址,接着执行下一跳指令;要么从出错地址开始执行,这就得由异常的性质种类决定了。

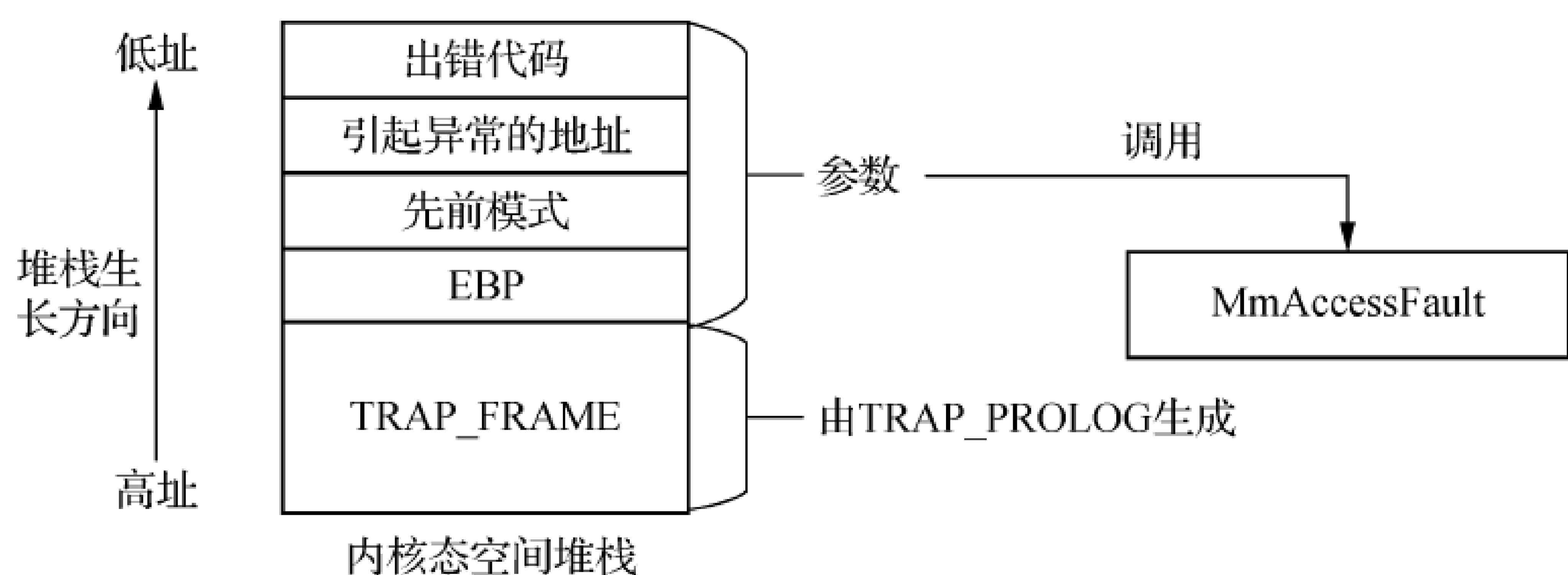


图 9 - 8　KiTrap14 形成的自陷框架和 MmAccessFault 的调用框架

➢ 如果在上述过程中异常未解决，则跳转到 AccessFail 指令模块处理真正的异常。首先调用异常处理的公共入口 CommonDispatchException，这个入口函数会先在内核态堆栈中构筑一个异常记录块 ExceptionRecord，用以记录本次异常的信息。

异常记录块数据结构中各个域如图 9 - 9 所示，其中：

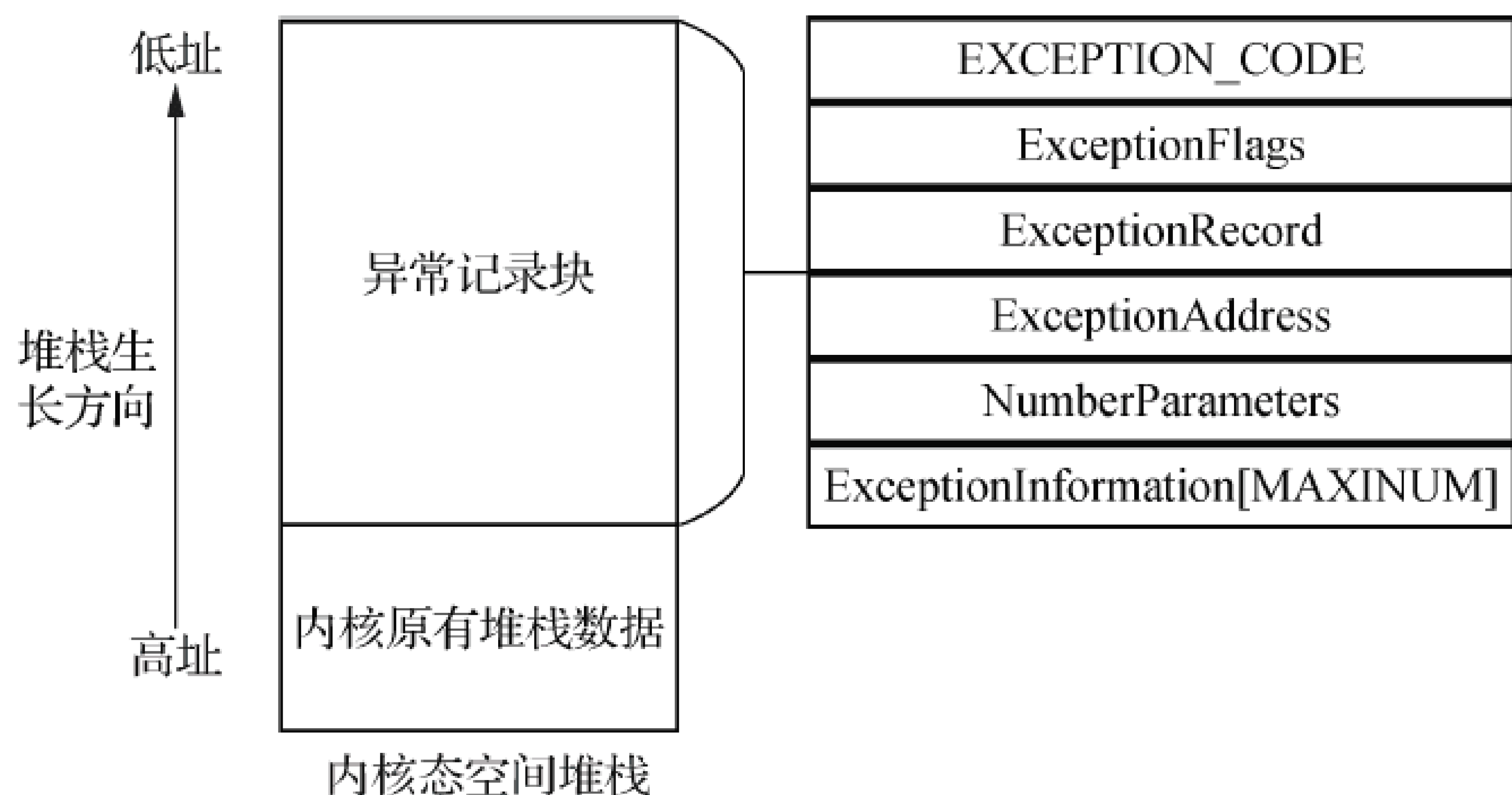


图 9 - 9　CommonDispatchException 构筑的异常记录块数据结构

➢ **EXCEPTION_CODE**：表示异常代码，例如 KI_EXCEPTION_ACCESS_VIOLATION 表示页面访问异常；

➢ **ExceptionRecord**：该指针指向上一个 ExceptionRecord 结构，即前一次的异常记录块；

➢ **ExceptionAddress**：表示本次异常结束后的返回地址，即自陷框架中的 EIP 指令指针，这是在执行序言 TRAP_PROLOG 的时候指定的，其实就是出错指令地址或者是其下一条指令地址；

➢ **ExceptionInformation**：记录了异常的其他扩展信息；

➢ **NumberParameters**：ExceptionInformation 的有效数据的项数。

构筑完异常记录块后，要调用内核态的异常处理的分发入口函数 KiDispatchException。只是在调用之前要将异常记录块的地址、先前模式（用户模式还是内核模式）和堆栈上的自陷框架指针等作为参数压入栈中。

下面来看 KiDispatchException 的执行流程，如图 9 - 10 所示，其执行步骤如下：

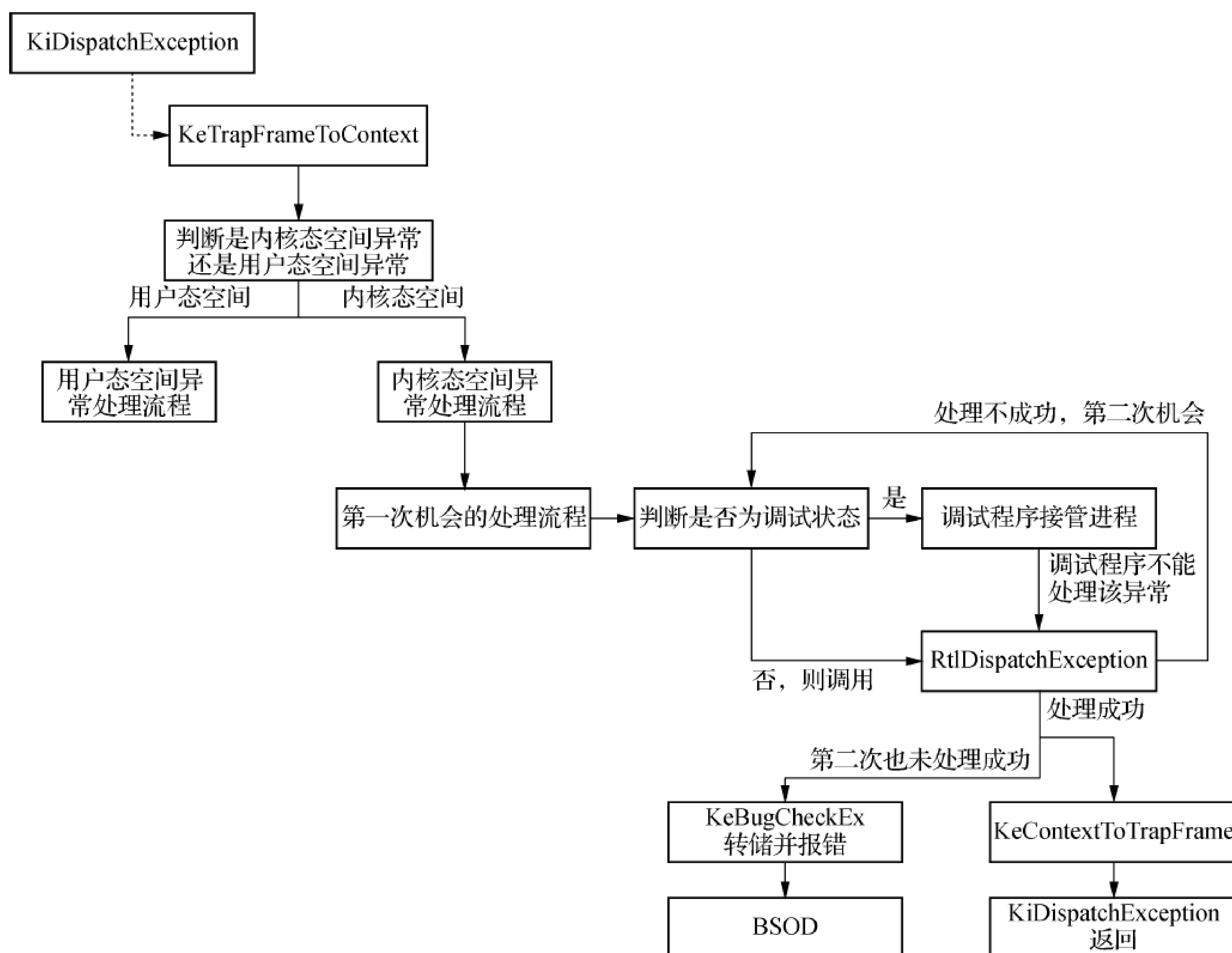


图 9-10　KiDispatchException 在内核态空间的执行流程

（1）KiDispatchException 首先调用 KeTrapFrameToContext 将内核态堆栈中的自陷框架转化成上下文结构 CONTEXT，并以 CONTEXT 为参数调用 KiDebugRoutine，判断当前是否为调试状态：

- 如果是调试状态，则由调试程序接管异常进程（前文有体现，调试程序优先级高于异常处理例程）；
- 如果不是调试状态，则调用 RtlDispatchException 遍历 ExceptionList 以找到能处理该异常的 SEH 框架来进行处理。

（2）如果 RtlDispatchException 处理成功，则意味着已经找到了能处理当前异常的 SEH 框架，继而调用 KeContextToTrapFrame 将传给 RtlDispatchException 的 CONTEXT 再转换回 TrapFrame。之所以要这样转来转去，是因为在异常处理的过程中 CONTEXT 内部的值可能会发生改变，例如 EIP 指令指针可能会变成出错指令的下一条，因此 CONTEXT 必须做一次更新。调用完成，处理函数也就返回了，至此本异常处理流程结束。

（3）如果 RtlDispatchException 处理不成功，则系统还会再给一次机会，会重新判断是否为调试状态，即调用 KiDebugRoutine。当第二次机会的处理流程走到 RtlDispatchException 时，能处理成功一切好说，不能处理成功则证明本异常"无可救药"了，应该执行

KeBugCheckEx 以显示出错信息并将现场转储为 Dump 文件。因为是在内核态出现的异常，通常会出现“死亡蓝屏”(Blue Screen Of Death,BSOD)。

从以上过程也可以看出，异常处理的核心就是对 RtlDispatchException 的调用，即从 ExceptionList 队列中找到能处理当前异常的 SEH 框架。对于 RtlDispatchException 的调用有三种结果：

- 该异常被某个 SEH 框架认领并实施长程跳转(_SEHCompilerSpecificHandler)，一旦实施了长程跳转，被跨越的那些 SEH 框架需要执行栈展开，因此长程跳转之前要先进行栈展开处理。一旦实施了长程跳转，RtlDispatchException 是不会返回的。
- 该异常被某个 SEH 框架认领，但不需要做长程跳转，只需要执行被越过 SEH 框架的善后函数，即栈展开函数，之后从 RtlDispatchException 返回以继续执行原指令。
- 所有 SEH 框架都拒绝认领该异常，则 RtlDispatchException 返回失败，接下来或者进行第二次尝试，或者直接出现 BSOD。

其实 RtlDispatchException 的执行过程非常简单：获取 ExceptionList 队列，从后向前遍历 SEH 框架(EXCEPTION_REGISTRATION_RECORD 结构，对应 SEH 的_SEHRegistration 结构)，针对每个框架执行 RtlpExecuteHandlerForException 来进行尝试。这里要注意 ExceptionList 中的每个节点是一个全局 SEH 框架，该框架可能是单独一个框架，也可能是一个框架栈，如果是框架栈则一定是个局部框架栈。

RtlpExecuteHandlerForException 可能有以下几种尝试结果：

- **不予认领继续执行**：此时应该向上回溯去尝试当前 SEH 框架的上一个框架，即更外层的 SEH 框架。
- **认领了但不需要做长程跳转**：这说明问题已经解决，该异常可以忽略，继续执行当前指令。
- **发生嵌套异常**：在异常处理过程中又发生了异常，此时要先处理后面发生的这个异常。

我们再来下钻一下 RtlpExecuteHandlerForException 的执行过程。

RtlpExecuteHandlerForException 将 RtlpExceptionProtector 函数指针赋值给 EDX 寄存器，并跳转到 RtlpExecuteHandler 执行，函数执行的参数与上一层的 RtlpExecuteHandlerForException 一致。不过如果不执行 RtlpExecuteHandlerForException 而是执行 RtlpExecuteHandlerForUnwind，那么 EDX 寄存器会被赋值为 RtlpUnwindProtector 并跳转到 RtlpExecuteHandler 执行，也就是说 RtlpExecuteHandler 有两种进入的场景，如图 9－11 所示。

RtlpExecuteHandler 这个函数最终要调用由 RtlpExecuteHandlerForException 传下来的参数——ExceptionHandler 函数，这是由用户设置的异常处理函数，因此 RtlpExecuteHandler 本质上还是对用户设置的异常处理函数的调用，对于一般类型的 SEH 框架这个处理函数就是 SEHFrameHandler。

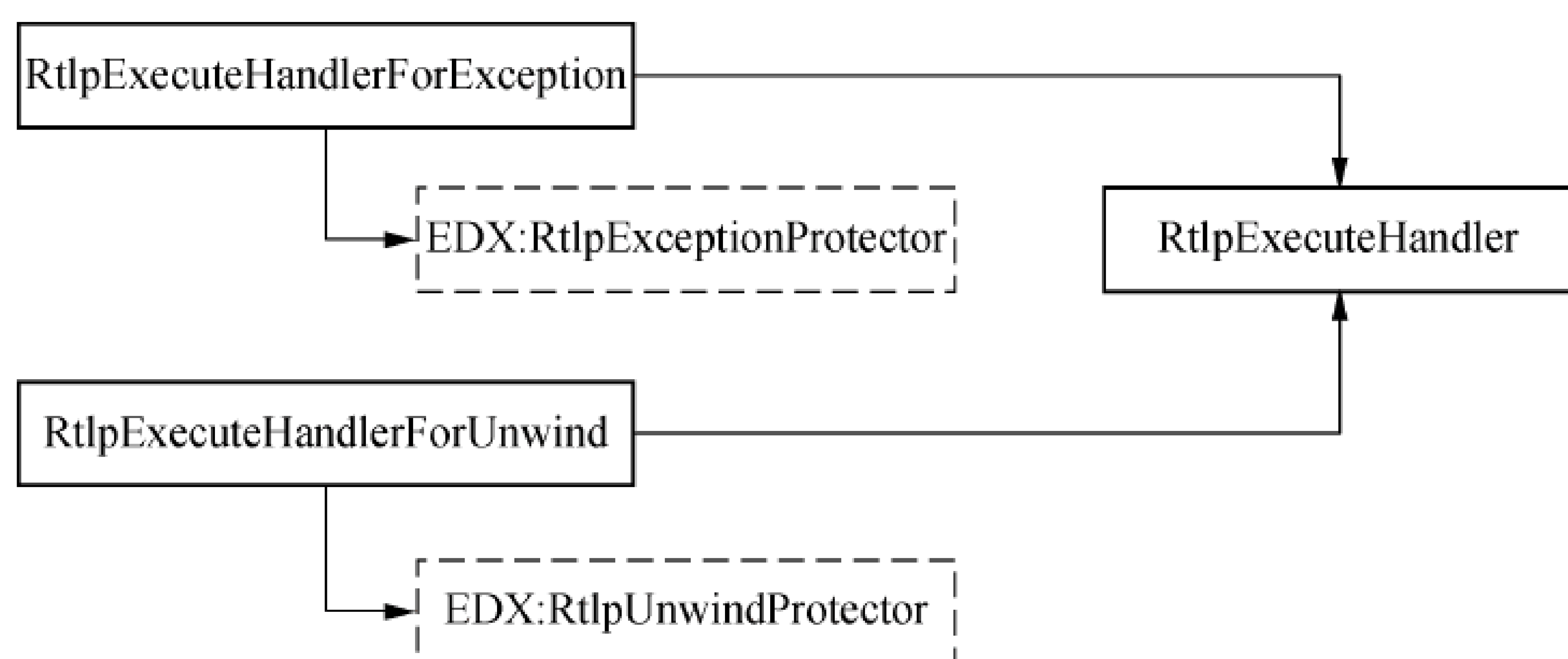


图 9－11　调用 RtlpExecuteHandler 的两种场景

这里要插播一下：RtlpExecuteHandler 会在内核态堆栈中生成一个异常处理“保护框架”，因为在调用 SEHFrameHandler 的过程中也有可能发生新的异常，即发生嵌套异常，前面的异常还没处理完呢，这又发生了异常，这是多么糟糕的事情。不过债多了不愁，既然又发生了异常那就先从新的当前异常开始处理吧，这个处理的依据就是保护框架，处理的路线还是从 RtlDispatchException 开始，这又是熟悉的套路了。当然，如果在执行 SEHFrameHandler 的过程中没有发生新的异常，那么这个保护框架也是做了一回有备无患的“带刀侍卫”，RtlpExecuteHandler 在返回的过程中会把它删除的。

“保护框架”的处理函数就是 RtlpExceptionProtector 或 RtlpUnwindProtector，但是它们并不实质处理新发生的异常，只是返回一个值表明发生了嵌套异常。

回到 SEHFrameHandler 函数，这才是真正处理异常的实体。SEHFrameHandler 首先会判断这是栈展开还是异常处理：

➢ 如果是栈展开，则执行 SEHLocalUnwind；

➢ 如果是异常处理，则遍历搜索能认领和处理本次异常的 SEH 框架。

注意，这里遍历搜索的是当前某个全局 SEH 框架的局部框架栈，直到搜索到符合的 SEH 框架或者搜索完成也没找到就结束遍历。遍历时执行当前局部框架的过滤函数，由过滤函数判断是否处理本次异常，处理本次异常的时候则执行 SEHCallHandler 函数。

SEHFrameHandler 的执行流程如图 9－12 所示。

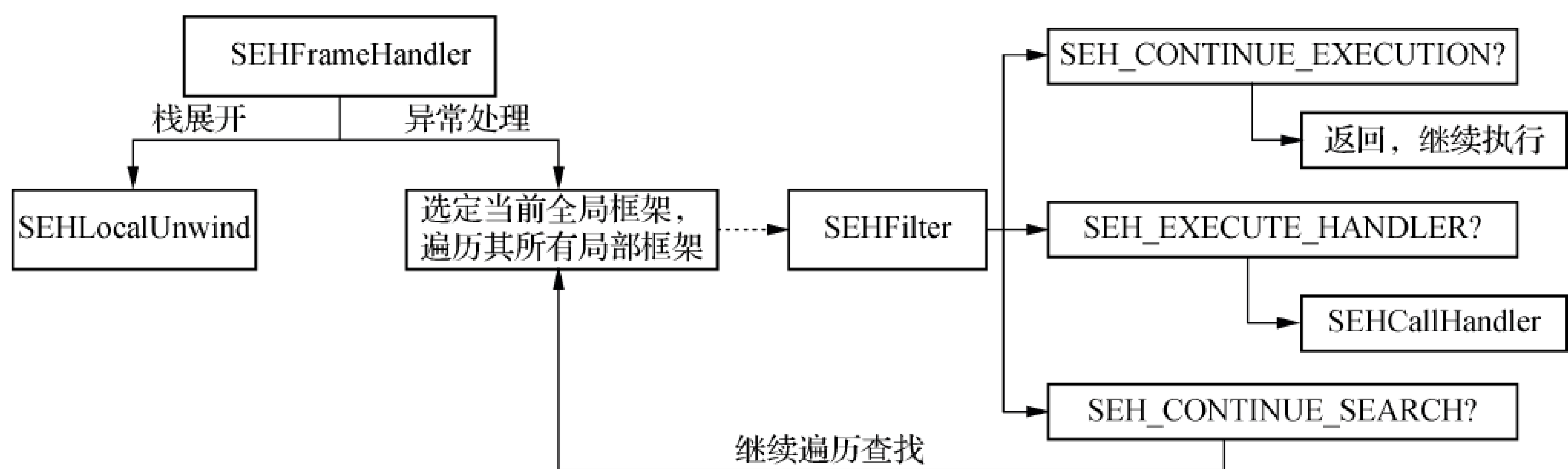


图 9－12　SEHFrameHandler 执行流程

SEHFilter 是当前的全局 SEH 框架中的局部 SEH 框架栈里每个局部 SEH 框架所携带的过滤函数。我们在前文说过，全局 SEH 框架在 ExceptionList 队列中，每个线程都有这样一个队列，这个队列中的每个元素都代表一个全局 SEH 框架，因为是全局的，或者说是由层层函数调用形成的，所以它们也会在堆栈中出现并形成栈链。局部 SEH 框架则不是由函数层层调用形成的，而是在一段代码中嵌套形成的，因此无法在堆栈中形成栈链，因而也就不能在 ExceptionList 中体现，只好采用另一个队列来存放，并且这个队列也是局部的，是每个全局 SEH 框架所独有的，如图 9－13 所示。当然也不是每个全局 SEH 框架都有局部 SEH 框架栈，有局部框架栈的还是少数，因为我们在使用"try-catch"框架时很少采用嵌套形式。

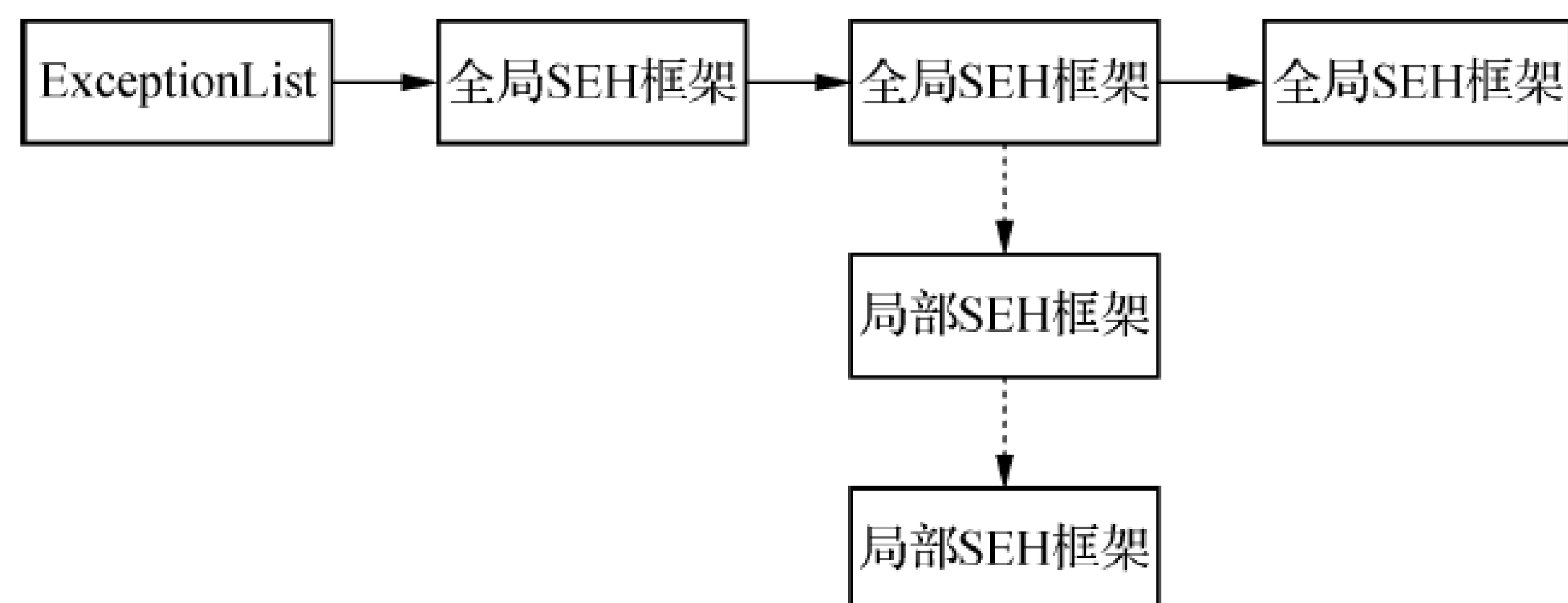


图 9－13　全局 SEH 框架与局部 SEH 框架

SEHCallHandler 则是真正实施长程跳转的函数，执行了这个函数，SEHFrameHandler 就不返回了。SEHCallHandler 的执行包含了全局栈展开、局部栈展开和长程跳转三部分：

- **执行 SEHGlobalUnwind 进行全局栈展开：** 所谓栈展开就是执行框架的善后函数以释放资源，善后函数就是前文提到的 RtlpExecuteHandlerForUnwind。在这一部分执行 ExceptionList 中位于当前选择的全局 SEH 框架之前的所有 SEH 框架的展开（资源释放）。在执行过滤函数的时候会发生框架跨越现象，这是由于前面的框架处理不了当前的异常，因此也只有在执行过滤函数的时候才有可能发生框架跨越，也才有可能执行栈展开函数。
- **执行 SEHLocalUnwind 进行局部栈展开：** 执行当前全局 SEH 框架的局部 SEH 框架队列之前的所有局部框架的展开（资源释放）。
- **执行框架的异常处理函数进行长程跳转：** 即执行 SEHCompilerSpecificHandler 函数，这个函数是不返回的。

9.3　用户态空间的结构化异常处理流程

前面的章节讲述了在内核态空间中的结构化异常处理流程，本节我们介绍在用户态空间的结构化异常处理。用户态空间的异常处理流程与内核态空间的大致相同，都是遍历 ExceptionList 以寻找能够处理当前异常的 SEH 框架。但是这里的 ExceptionList 与内核态空间所说的 ExceptionList 不是同一个。内核态空间的 ExceptionList 是个全局列表，是 KPCR 数据结构中第一个域（线程信息块 NT_TIB 结构）的第一个元素所指向的 ExceptionList，并且由

于是在内核态空间,该列表全局只有一个(每个处理器有一个),如图 9－14 所示。

```
kd> dt _KPCR
nt!_KPCR
  +0x000 NtTib          : _NT_TIB
  +0x01c SelfPcr        : Ptr32_KPCR
  +0x020 Prcb           : Ptr32_KPRCB
  +0x024 Irql           : UChar
  +0x028 IRR            : Uint4B
  +0x02c IrrActive      : Uint4B
  +0x030 IDR            : Uint4B
```

```
kd>DT _NT_TIB
nt!_NT_TIB
  +0x000 ExceptionList   : Ptr32 _EXCEPTION_REGISTRATION_RECORD
  +0x004 StackBase       : Ptr32 Void
  +0x008 StackLimit      : Ptr32 Void
  +0x00c SubSystemTib    : Ptr32 Void
  +0x010 FiberData       : Ptr32 Void
  +0x010 Version         : Uint4B
  +0x014 ArbitraryUserPointer : Ptr32 Void
```

图 9－14　KPCR 数据结构中的 ExceptionList

而用户态空间的 ExceptionList 则是 TEB 数据结构(如图 9－15 所示)中第一个域(NT_TIB 结构)的第一个元素所指向的列表,是在用户态空间的队列,并且每个线程都有一个这样的队列。在系统中有多少个用户态线程,就有多少个这样的队列。

```
typedef struct _TEB
{
     NT_TIB NtTib;
     PVOID EnvironmentPointer;
     CLIENT_ID ClientId;
     PVOID ActiveRpcHandle;
     PVOID ThreadLocalStoragePointer;
     PPEB ProcessEnvironmentBlock;
     ULONG LastErrorValue;
     ULONG CountOfOwnedCriticalSections;
```

图 9－15　TEB 数据结构

FS 寄存器在用户态空间指向当前线程的 TEB 结构,在内核态空间则指向 KPCR 结构,而且 ExceptionList 都是 TEB 或者 KPCR 数据结构的第一个域的第一个元素,因此无论是在用户态空间还是内核态空间,获取 ExceptionList 的方式都是一样的,这也为结构化异常处理提供了很大的便利。

前文讲过,当异常发生时,内核异常分发例程首先根据异常类型调用诸如 KiTrapX 这样的函数,KiTrapX 会先判断这些异常是真异常还是假异常并进行相应处理(例如 MmAccessFault 处理内存换页异常),如果处理完了异常还未解决,那就证明这是真的异常了,就会调用异常处理的公共入口函数 CommonDispatchException,再进入 KiDispatchException 进行实质异常处理。处理也分为用户态的和内核态的,并且处理流程就此分叉。针对用户态的异常处理的主要思想也是给予两次尝试机会。

(1) 第一次尝试机会

➢ 调用 KiDebugRoutine 交由内核态空间调试程序处理。

- 如果存在内核调试程序并且调试程序也解决了本次异常,则跳转到 KeContextToTrapFrame 函数,并将上下文 CONTEXT 转换为自陷框架 TrapFrame(更新发生异常时用户态空间某些寄存器的值),函数 KiDispatchException 返回。
- 如果不存在内核调试程序或调试程序未能解决本次异常,会转而尝试调用用户态空间调试程序 DbgkForwardException。如果用户态空间调试程序能解决异常,则

KiDispatchException 返回;若不能解决该异常,则开始调用用户态空间的异常处理例程——调用用户态空间的 SEH。

➢ 在调用用户态空间 SEH 之前先设置好用户态空间堆栈,为回到用户态空间去处理异常做准备。这里要注意,我们处理异常的时候一直都在内核态执行操作,只有需要在用户态处理异常的时候,才会像 APC 处理一样先安排好用户态空间堆栈,再伺机返回用户态空间。
   - 首先在用户态空间堆栈构筑一个 CONTEXT 数据结构和一个 EXCEPTION_RECORD 数据结构,并将这两个数据结构的地址压入用户态堆栈,如图 9－16 所示。其中,CONTEXT 即为之前通过 KeTrapFrameToContext 生成的数据结构,表示异常发生时的各寄存器值;EXCEPTION_RECORD 是从 KiDispatchException 传下来的参数,表示当前异常的记录块。

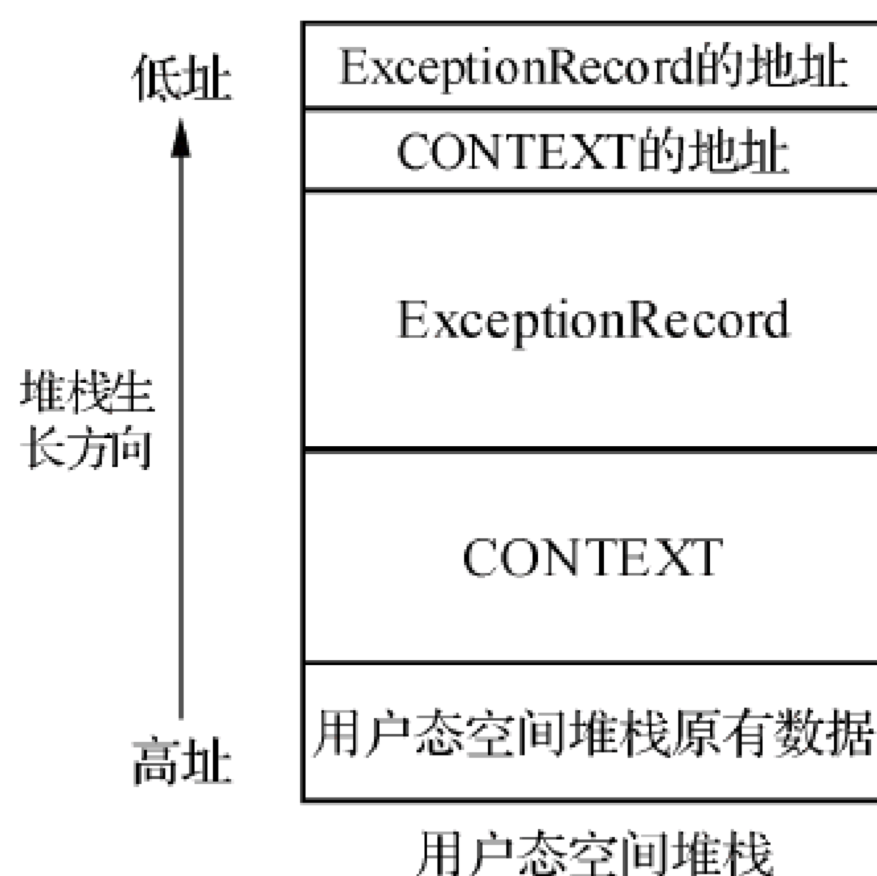


图 9－16　第一次尝试中对用户态空间堆栈的修改

   - 完成了用户态空间堆栈的修改后,将新的堆栈指针的值(ExceptionRecord 地址所在位置)和堆栈基址(KGDT_R3_DATA)写入内核态空间堆栈的自陷框架中,KiSSToTrapFrame、KiEspToTrapFrame 这两个函数分别完成自陷框架 SS 和 ESP 位置的更新写入。
   - 将内核态堆栈自陷框架中用户态空间返回地址(EIP 位置)更改为 ntdll. dll 中的 KeUserExceptionDispatcher,这是用户态空间 SEH 的总入口,也是为回到用户态空间做的最重要的一步准备,之前在用户态空间堆栈安排的两个数据结构 CONTEXT 和 ExceptionRecord 是该总入口函数的参数。

至此,返回用户态空间的准备工作就完成了,线程返回用户态空间,一返回就开始执行事先铺设的 KeUserExceptionDispatcher。KeUserExceptionDispatcher 遍历 TEB 中的 ExceptionList。如果有能够处理当前异常的 SEH 框架则处理当前异常;若 ExceptionList 中也没有能处理该异常的 SEH 框架,那么就要执行第二次尝试了:调用 ZwRaiseException 引起一次“软异常”,这是故意由软件方式引起的异常,从而使之再次进入异常处理的公共入口 CommonDispatchException。

(2) 第二次尝试机会

从 CommonDispatchException 进来后,由于是“二进宫”,再次调用 DbgkForwardException,

尝试用户态空间调试程序是否能够处理当前异常：

- 如果能处理，则万事皆休，KiDispatchException 返回；
- 如果不能处理，则调用 ZwTerminateProcess 结束当前进程，并且通知环境子系统进程 csrss. exe 出错，最后调用 KeBugCheckEx 显示进程出错信息(如图 9－17 所示)，并转储异常发生时的内存镜像。

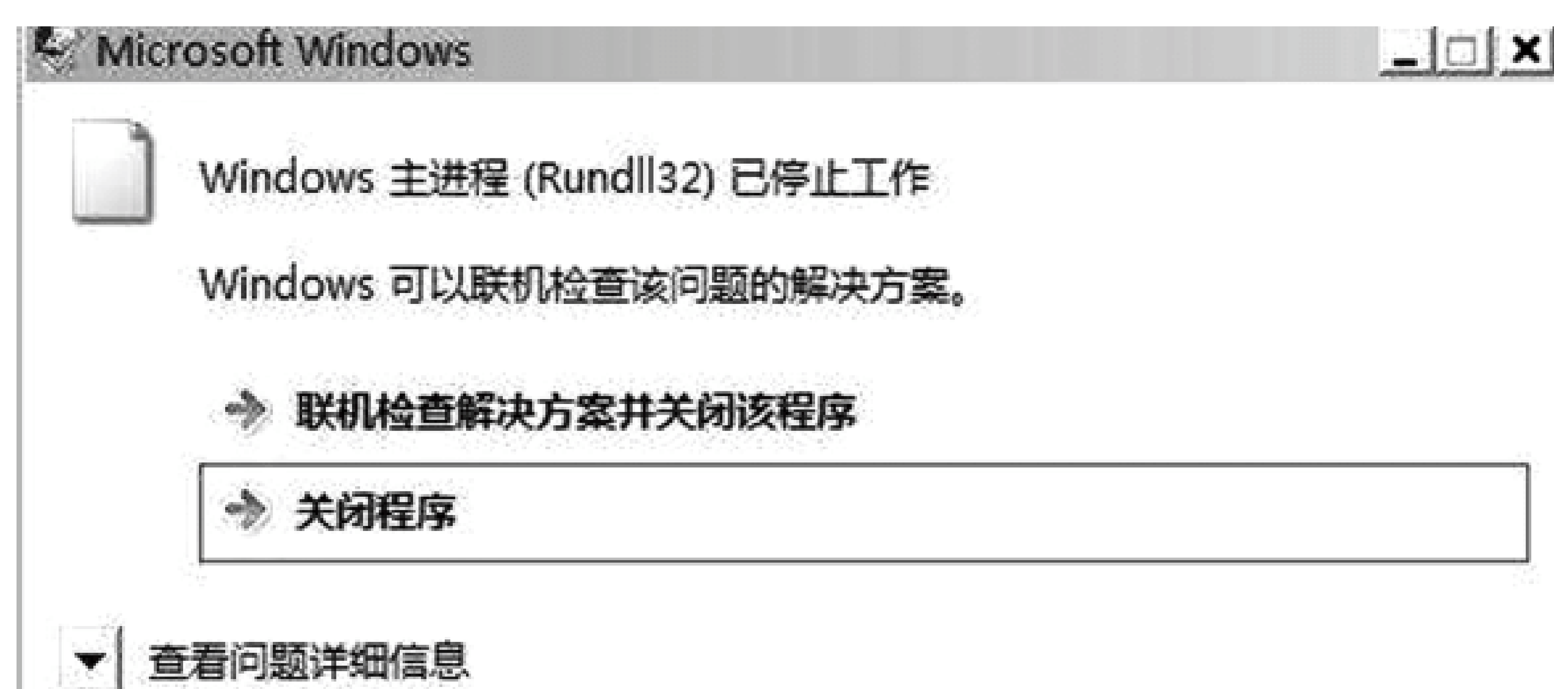


图 9－17　csrss. exe 弹出的错误提示框

图 9－18 是 KiDispatchException 在用户态空间的执行流程，我们可以对比其在内核空间的执行流程来理解。

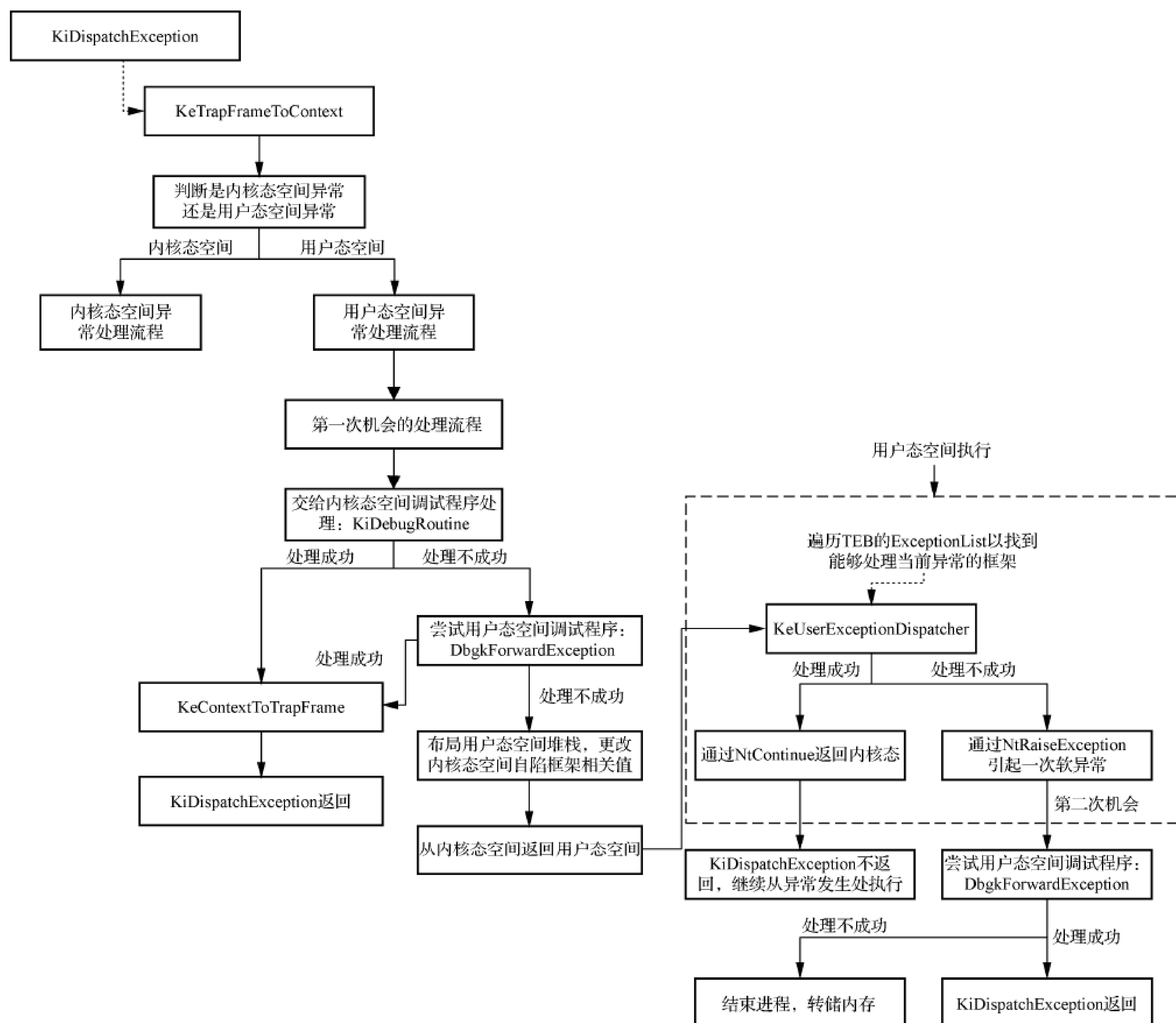


图 9－18　KiDispatchException 在用户态空间的执行流程

下面介绍一下向量化异常处理机制。

KeUserExceptionDispatcher 作为用户态空间 SEH 的总入口，首先会执行向量式异常处理（VEH）框架：RtlpExecuteVectoredExceptionHandlers。这种异常处理框架在每个进程中都有一个，挂载在 RtlpVectoredExceptionHead 链表中，这个链表的元素结构是这样的：

```
typedef struct _VECTORED_LIST                              //链表结构
{
    _VECTORED_LIST * pNext;                                //链表的向后指针
    _VECTORED_LIST * pPrevious;                            //链表的向前指针
    DWORD dwMark;                                          //一个有效标记
    PVECTORED_EXCEPTION_HANDLER VectoredFunction;          //加密后的异常处理函数指针
}VECTORED_LIST, * PVECTORED_LIST
```

当 PEB 的 EnvironmentUpdateCount 标志位为 0 的时候，异常发生时会检查 VEH 链表并执行向量化异常处理；否则，VEH 机制是失效的。

VEH 是基于进程的异常处理机制，使用 Windows API（AddVectoredExceptionHandler）进行注册，可以进行多次注册以形成 VEH 结构体双向链表（SEH 结构体链表是单向的），这些 VEH 结构体保存在进程堆中（SEH 结构体保存于栈中），而且注册时可以自由指定 VEH 结构体在链表中的位置，而不是后注册的先执行，因此将异常处理的优先权完全交给了开发者。更重要的是 VEH 的优先级高于 SEH（但仍然低于调试器处理的优先级），并且 VEH 可以在 SEH 之前执行，也可以在 SEH 之后执行，这一点用户可以自由定义。与 SEH 相比，VEH 没有栈展开机制，善后工作不需要用户自己定义。

VEH 有两个链表头：链表 0 和链表 1，前者的异常处理框架是由函数 AddVectoredExceptionHandler 挂载上去的（Windows XP 及以上的版本支持），后者的异常处理框架则是由函数 AddVectoredContinueHandler 挂载上去的（Windows Vista 及以上的版本支持），分别通过 RemoveVectoredExceptionHandler 和 RemoveVectoredContinueHandler 进行框架删除。无论执行在前还是在后，只要前者的异常处理框架处理了当前的异常，就不再分发给后面的异常处理框架了，由此可见 VEH 给了应用程序一个先于 SEH 框架处理异常的手段。

最后提一下软异常。所谓软异常就是由软件触发的异常，这是一种以主动方式触发的异常，用于再次回到异常处理框架中，C++ 异常处理框架的 throw 关键字就用于对软异常的调用。软异常的调用函数有两个：RtlRaiseException 和 RtlRaiseStatus。前者的参数是个异常记录块 EXCEPTION_RECORD，后者的参数是异常的出错状态码，但两者都是基于系统调用 NtRaiseException 的。

NtRaiseException 的执行流程如图 9 – 19 所示。NtRaiseException 首先调用 KiRaiseException 以将作为参数传下来的上下文结构 CONTEXT 转换成自陷框架 TrapFrame，并调用 KiDispatchException。对于 KiDispatchException 我们就很熟悉了，无论是在内核态空间还是用户态空间，都遍历 ExceptionList 并从 KiRaiseException 返回。之后根据 KiRaiseException 返回的状态码（表示处理异常是否成功）来决定是调用 KiServiceExit（处理异常不成功）还是 KiServiceExit2（处理异常成功）。KiServiceExit 和 KiServiceExit2 的内部处理流程一致，但参数不一样，特别是对于尾声 TRAP_EPILOG 的处理。

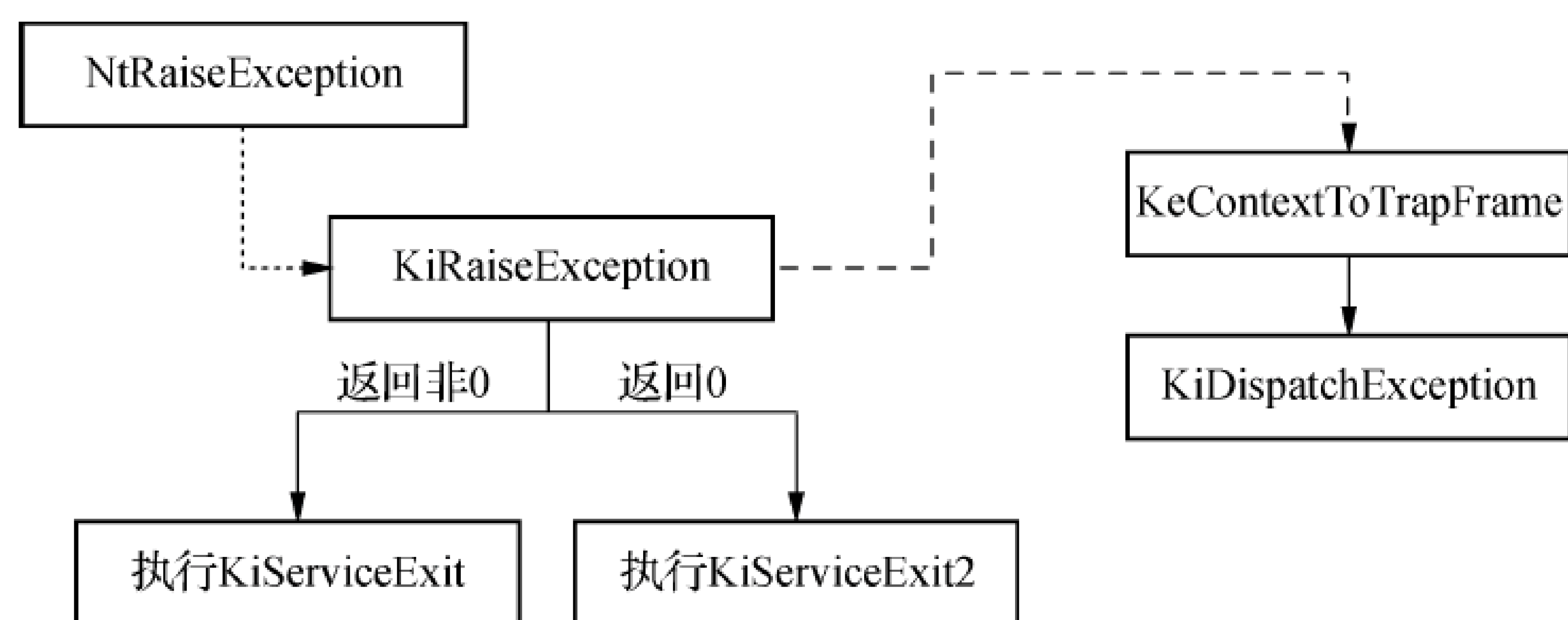


图 9－19　NtRaiseException 的执行流程

本章小结

本章以堆栈动态平衡为主线介绍了 Windows 异常处理的相关流程，包括内核态空间的异常处理和用户态空间的异常处理。最后介绍了什么是向量化异常处理。

第10章　内存管理机制

内存管理是操作系统的核心模块,本章将按照图 10-1 所示的提纲进行详细介绍。

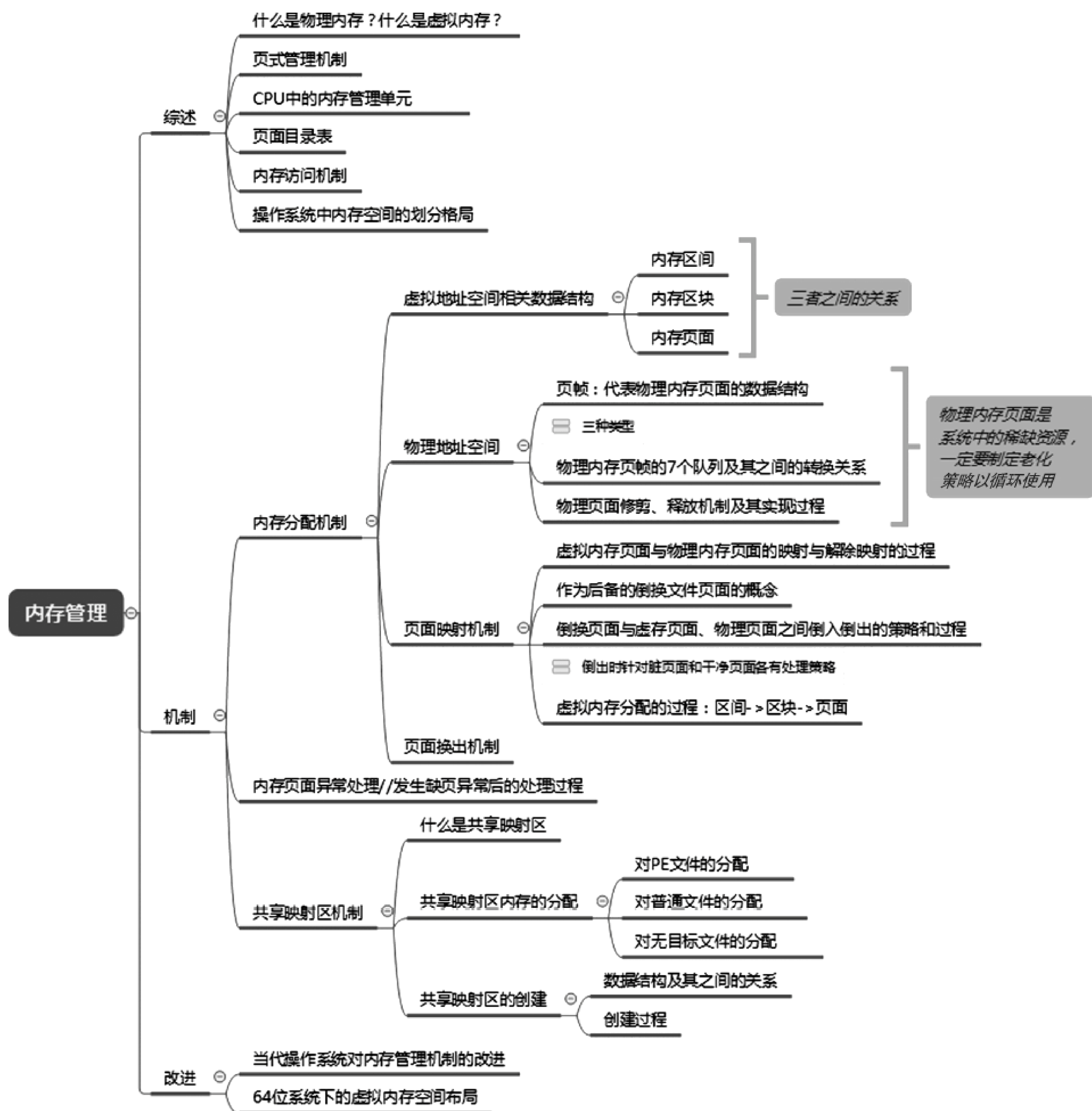


图 10-1　本章提纲

10.1　内存管理综述

1. 物理内存与虚拟内存

内存管理是所有操作系统的核心功能模块,在现代 Windows 操作系统中是采用虚拟地

址+物理地址的方式来管理内存的。例如我们常说"电脑有 4 GB 的内存",这是指物理内存;说"进程地址访问空间 4 GB",这是指虚拟内存。

那为什么不简单地采用直接物理地址的管理方式呢?我们试想一下两个进程都在一个物理内存空间中运行,起始地址都一样,那势必有一个进程要让步,要调整加载的起始地址做"重定向",否则就会产生地址争用问题。而现代操作系统是多任务系统,并且所有进程的起始地址都一样,那么如果采用直接物理地址的管理方式,可以想象系统中的各进程要具备怎样复杂的协调机制才能保证地址不被争用。

在一个物理内存空间的各个进程就算解决了"重定向"问题,一个进程运行难免不会打扰别的进程运行,例如一个越界访问,可能就把另一个进程的内存破坏了。而进程是由人开发的,难免会出现这样的漏洞,因此进程地址空间的隔离非常重要,而且只能把隔离机制交给操作系统实现,这也客观上要求把进程能访问的地址空间虚拟化,在这个虚拟空间内进程"天马行空,唯我独尊",而对物理内存的管理使用则交给操作系统。

因此由操作系统自己来完成各进程的重定向以解决内存地址争用问题既是给应用进程"减负",也是操作系统的必然选择。Windows 系统采用虚拟地址和物理地址相结合的方式,进程能看到的只有虚拟地址,每个虚拟地址空间的大小都是 4 GB(32 位系统下),在这 4 GB 地址空间中只运行这一个进程,因此不存在地址重定向问题,而虚拟地址到物理地址的映射翻译由操作系统完成。

2. 页式管理

同时,Windows 内存管理采用了页式管理机制。所谓页式管理就是将内存(包括物理内存和虚拟内存)、磁盘等存储介质按照一定的大小(例如 4 KB)分割成一页一页的内存页面或磁盘页面,磁盘上的数据以页面(Page)为单位加载到物理内存,物理内存也以页面为单位换入、换出这些数据,同时虚拟内存更是以页面为单位与物理内存进行映射。访问一个虚拟内存页面必然对应映射了一个物理内存页面,毕竟只有物理内存才能承载虚拟内存中的内容。页式管理使内存管理简便化了,访问一个地址的时候要将被访问内容加载到内存中,但不只是待访问的一小块数据被加载到内存,而是该地址所在的整个内存页面被加载到内存。在 32 位系统下,整个虚拟地址空间的大小是 4 GB,一个页面的大小是 4 KB,因此虚拟内存可以划分为 1 M 个页面。物理内存也一样,例如系统中只有 2 GB 内存,则可以划分为 512 K 个内存页面。每个页面的起始地址必然是 4 KB 对齐的(所谓对齐,就是起始地址是 4 KB 的整数倍)。这样做的好处是:

- 32 位系统下只管理 1M 个页面,管理数量和范围可控。
- 对于页面的管理以一个页面为步长,这样做符合计算机的思考与读取习惯。
- 一般来说虚拟内存空间大于物理内存空间,在多进程环境下,借用分页和倒换机制,可以平衡内存的访问,"三个锅两个盖"使得物理内存总可以应付得了虚拟内存的资源需求。

3. 内存管理单元

CPU 由运算逻辑单元(Arithmetic Logic Unit,ALU)、内存管理单元(Memory Management Unit,

MMU)和 L1 Cache、L2 Cache 等组成。ALU 将指令或数据的虚拟地址发送给 MMU,MMU 将虚拟地址翻译为物理地址供 ALU 访问,并且也对地址的访问做权限检查。MMU 查找物理地址时首先在 TLB(快表)中查找页目录项,如果命中则直接从 TLB 中取目录项(页面地址),未命中则再从内存中获取目录项,从 TLB 中查找要比从内存中取址快得多。如图 10-2 所示,可以看出,MMU 在地址翻译中起到核心作用。

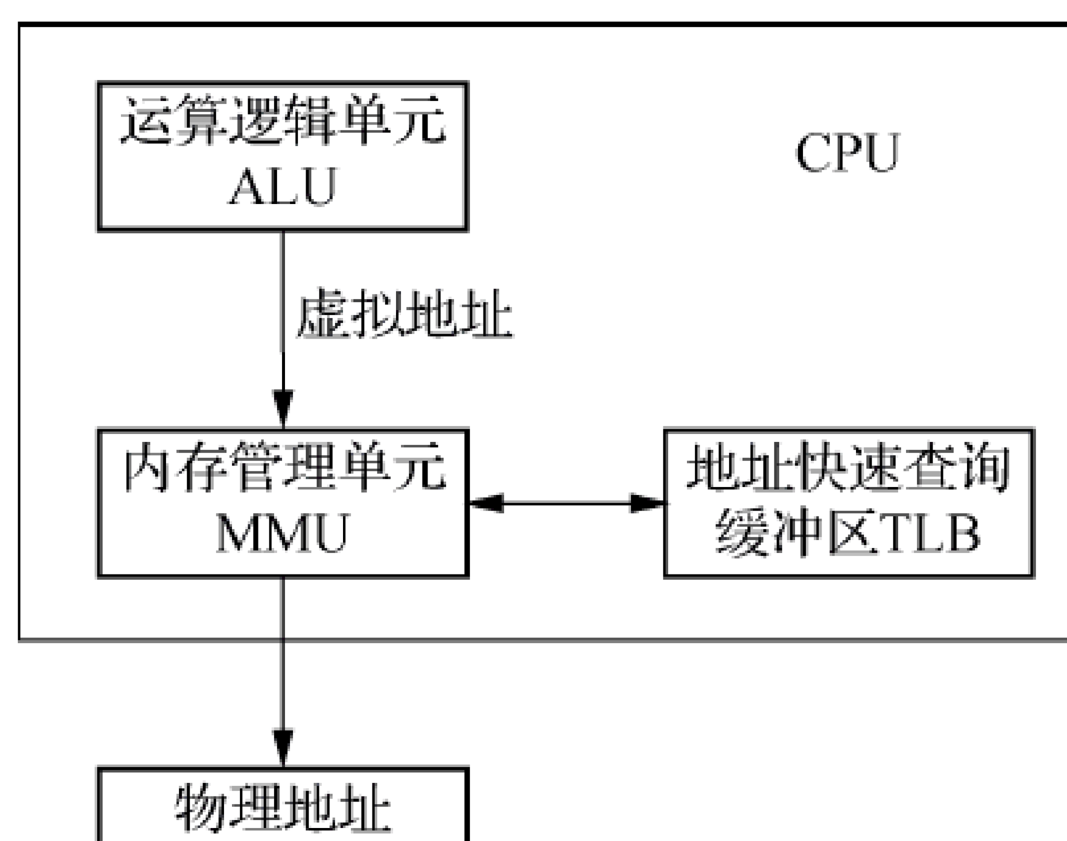


图 10-2　CPU 逻辑示意图

4. 页面目录表

CR3 寄存器存放了当前进程的页面目录表的物理地址,这个表是专门为进程访问内存页面而设计的。它是个二级表:将 4 GB 的虚拟地址空间划分为 1 M 个内存页。一维线性管理 1 M 个页面地址的开销也很大,但如果按照二级目录的方式组织对这 1 M 个页面的访问(采用二维数组的方式管理),则可以划分成 1 K 个一级索引,每个索引管理 1 K 个页面(二级索引),这种方式还是比较容易接受的,毕竟查找两次长度不是很长的数组也不是什么开销大得不得了的事。更重要的是,进程对内存页面的访问基本上只是访问几个页面,对应的页目录项也就是几个,如果做成一维数组,数组中基本上是页帧的空指针,但数组仍然要占据一个 4 MB 的内存空间(4 B×1 M),太不经济;而做成二维数组,这几个页面对应的页帧指针充其量也就占用几个页面,加上页目录项所在的页面也没有多少,对内存的需求要比一维数组集约很多。

在上述方案中,我们将一级索引称为页目录项(Page Directory Entry,PDE),将二级索引称为页表项(Page Table Entry,PTE)。每个 PDE 中存放了对应 PTE 表的物理内存页面的地址,一个 PDE 包含了 1K 个连续的 PTE;而一个 PTE 也代表了一个物理内存页面的地址,因此一个 PTE 表代表了 1 K 个物理内存页面,如图 10-3 所示。无论是 PDE 还是 PTE,在 32 位系统中都占用 4 个字节,因此 PDE 表占用一个 4 KB 内存页面(4 B×1 K),而一个 PTE 表也占用了一个 4 KB 内存页面。

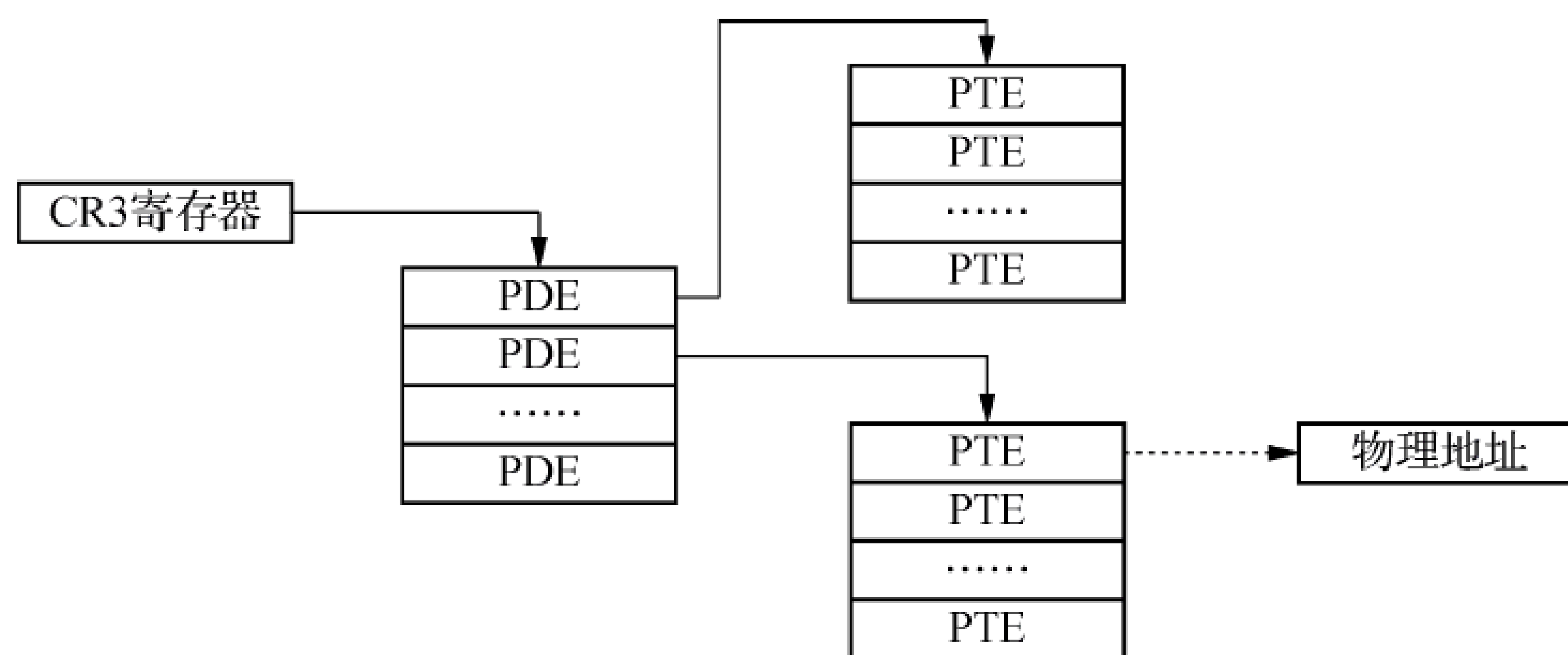


图 10-3　页面目录表结构

5. 内存访问

CPU 对于内存的访问过程大致如下：

（1）ALU 将指令中需要的虚拟地址发送给 MMU。

（2）MMU 根据虚拟地址的高 10 位和中间 10 位向 TLB 查询当前进程需要访问的页表项是否已经存在。在 32 位系统中使用高 10 位描述 PDE 的索引（一级索引），使用中间 10 位描述 PTE 的索引，使用低 12 位描述该虚拟地址所指向的内容在 PTE 所代表的内存页面的偏移。若存在，则向 TLB 获取 PTE 以返回给 ALU 并跳转到步骤（6）；否则，跳转到步骤（3）。虚拟地址与页面地址的映射关系如图 10－4 所示

（3）将当前进程的页面目录表物理地址赋值给 CR3 寄存器，该地址存放于当前进程的 KPROCESS 结构的 DirectoryTableBase 字段中。

（4）根据 CR3 描述的地址获取当前进程的页面目录表。

（5）在页面目录表中根据虚拟地址的前 20 位找到对应的 PDE 和 PTE，并刷新缓存至 TLB 中，同时将其返回给 ALU。

（6）ALU 根据获得的 PTE 访问内存页面，并将内存页面的内容刷新驻留到 Cache 中。

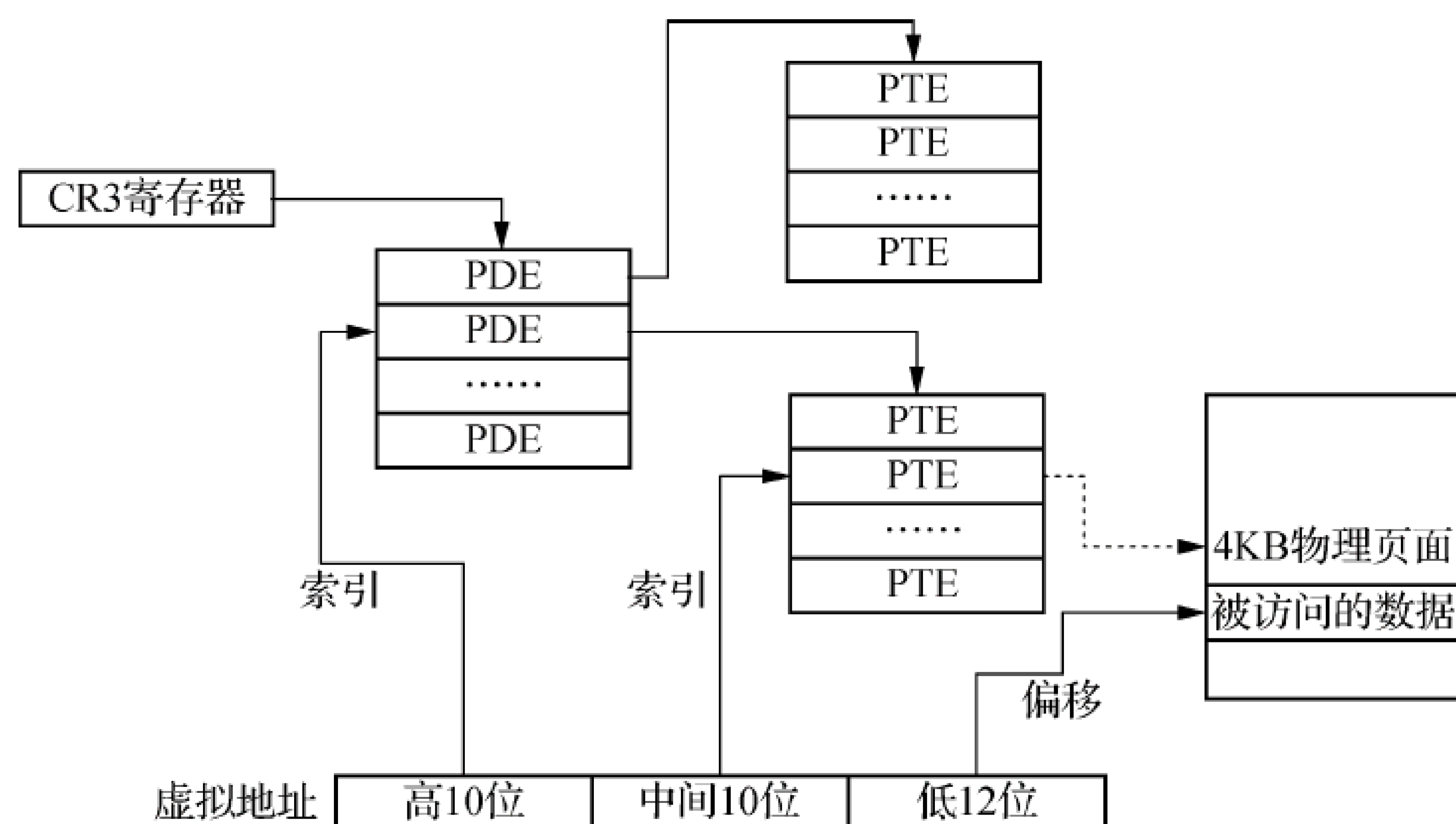


图 10－4　虚拟地址与页面目录表的映射关系

在进程中，对内存的访问带有一定的局部性，也就是说对于数据或代码的访问一般都集中在当前访问地址的周围，很少出现乱跳一气的现象（代码一般是顺序执行的，因此其在内存中的位置一般是相邻的，数据也同样带有类似的局部性，当然，使用了代码混淆加扰技术的除外）。同时，被访问的物理页面很有可能在短时间内再次被访问。采用分页机制后，我们在一段时间内访问的代码或数据大概率是在一个页面中，因此一个页面在内存中可能会使用很长时间而不被换出（页面内容一直被访问）。在一个进程的生命周期中，被访问的页面可能也就是十几个，而这十几个页面也许就分布在两三个 PTE 表中，再加上 PDE 表，至多也就占用三四个内存页。将少数的几个页面地址及其关系存放在高速访问缓存中有利于加快访问速度，减少内存访问，这也是设置 TLB 的初衷。

因此，进程被访问时，MMU 先探查 TLB 中是否存在要被访问页面的 PDE 与 PTE，如果

存在,MMU 就直接从 TLB 中读取,否则就从内存中读取页面目录表并将待访问的 PDE 和 PTE 调入 TLB 中以便下次访问。当然 TLB 也不是无限容量的,其缓存的页表项也在不断地老化,当页表项长时间没有被访问,其在 TLB 中也就逐渐老化失效了,可以把存储空间让给别的页表项,待下次再访问该页表项时重新调入就行了。

现代内存管理也支持页面倒换机制。前文说过,一般物理内存会小于虚拟内存,例如在 32 位系统中,虚拟内存空间为 4 GB,而物理内存空间可能只有 2 GB。在进程开得很多的情况下物理内存的需求也会变得很大,再加上系统自带的服务也需要很大数量的物理内存,就显得物理内存资源不够用了。物理内存不够用,系统响应就会变慢变卡,这是操作系统运行时需要竭力避免的。页面倒换机制支持将暂时不用的内存页面写回磁盘以释放对物理内存页面的占用,释放的物理内存页面被调度给其他进程使用。后文也会对这种机制进行详细描述。

6. 地址空间的划分

虚拟地址空间被分成了内核态空间和用户态空间两部分,在 32 位系统中以 0x80000000 为界,以上为内核态空间,以下为用户态空间,当然也可以通过配置将这个分界地址抬高到 0xC0000000,如图 10－5 右半部分所示。这样划分有利于实现内核态空间数据的共享和用户态空间数据的隔离。不过,在 32 位系统中也不是严格按照上述两个地址分界的,在这两个地址的上下以及周围还有一些“飞地”,代表了两个空间的缓冲隔离地带,因此也都是不可访问的。

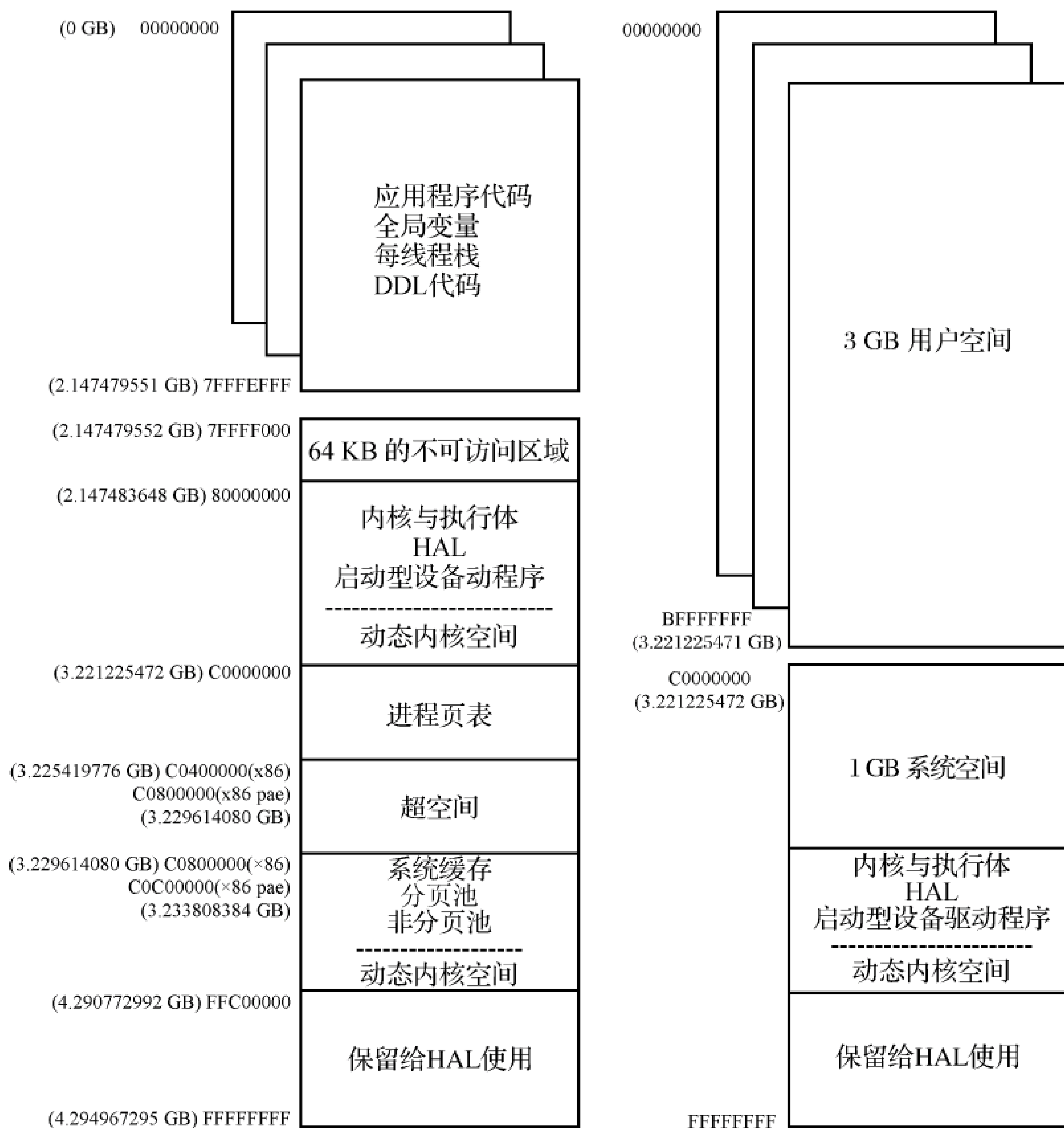


图 10－5　32 位系统中的两种虚拟地址空间布局

用户态空间的内存消费者一般有这样几类：Windows PE 文件映像、数据映射文件、堆栈，而按照状态来分又可分为 FREE(释放)、RESERVE(保留)、COMMIT(提交)三种。

综上所述，虚拟内存、物理内存、页面倒换是构成内存管理机制的核心。在 Windows 系统中，上述内存管理机制可以实现以下目标：

- 使现有的物理内存能够满足系统中各种进程的访问需要。
- 隔离各个进程的内存地址访问空间，使进程之间的运行互不干扰。
- 加速对内存的访问。
- 实现内存保护机制，例如只读内存不可写、可读写内存不能执行等，这在防止堆栈溢出漏洞中非常有意义。

10.2 内存分配机制

10.2.1 虚拟地址空间

在进程的执行体结构 EPROCESS 中存在描述用户态地址空间的数据结构，即 EPROCESS 的 VadRoot 域指向的 MADDRESS_SPACE 数据结构，而 MADDRESS_SPACE 的第一个域就是 MEMORY_AREA 数据结构。这两个数据结构中，前者叫作内存空间，后者包含了用户态地址空间中的地址区段、区块队列、起止地址等重要信息，叫作内存区间，如图 10－6 所示。

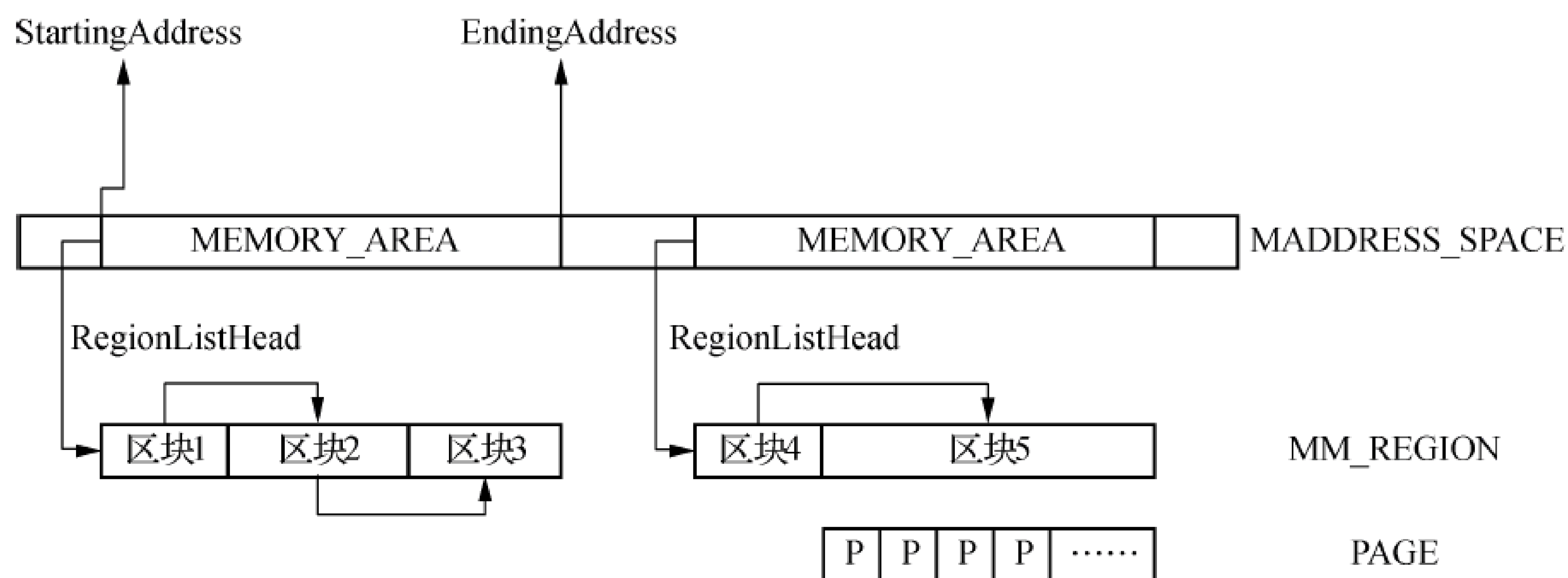


图 10－6 用户态地址空间的数据结构

我们首先来看看内存区间、内存区块和内存页面之间的关系。

1. 内存区间

内存空间的这 2 GB(或 3 GB)不是都在使用，可能只使用了其中一部分数据区间或代码区间，这些区间用 MEMORY_AREA 结构表示，它们是虚拟地址空间中已分配的区域，或者说就是使用 VirtualAlloc 函数 RESERVE(保留)或 COMMIT(提交)后的虚拟内存。

所谓预定就是分配区间但不映射虚拟内存页面，所谓提交就是既分配区间也映射虚拟内存页面。因此预定只能说明“此路是我开”，这段地址范围“我占了”，但并不能说明与虚

拟内存的映射也建立了,更不能说明用于承载内容的物理内存也分配了。

MEMORY_AREA 的组织形态是二叉树,如图 10 - 7 所示。左边的子树表示地址空间低于自己的区间范围,即 MEMORY_AREA2. EndingAddress < MEMORY_AREA1. StartingAddress;右边的子树表示地址空间高于自己的区间范围,即 MEMORY_AREA1. EndingAddress < MEMORY_AREA3. StartingAddress。以这样一种方式组织内存区间可以避免遍历式查找。

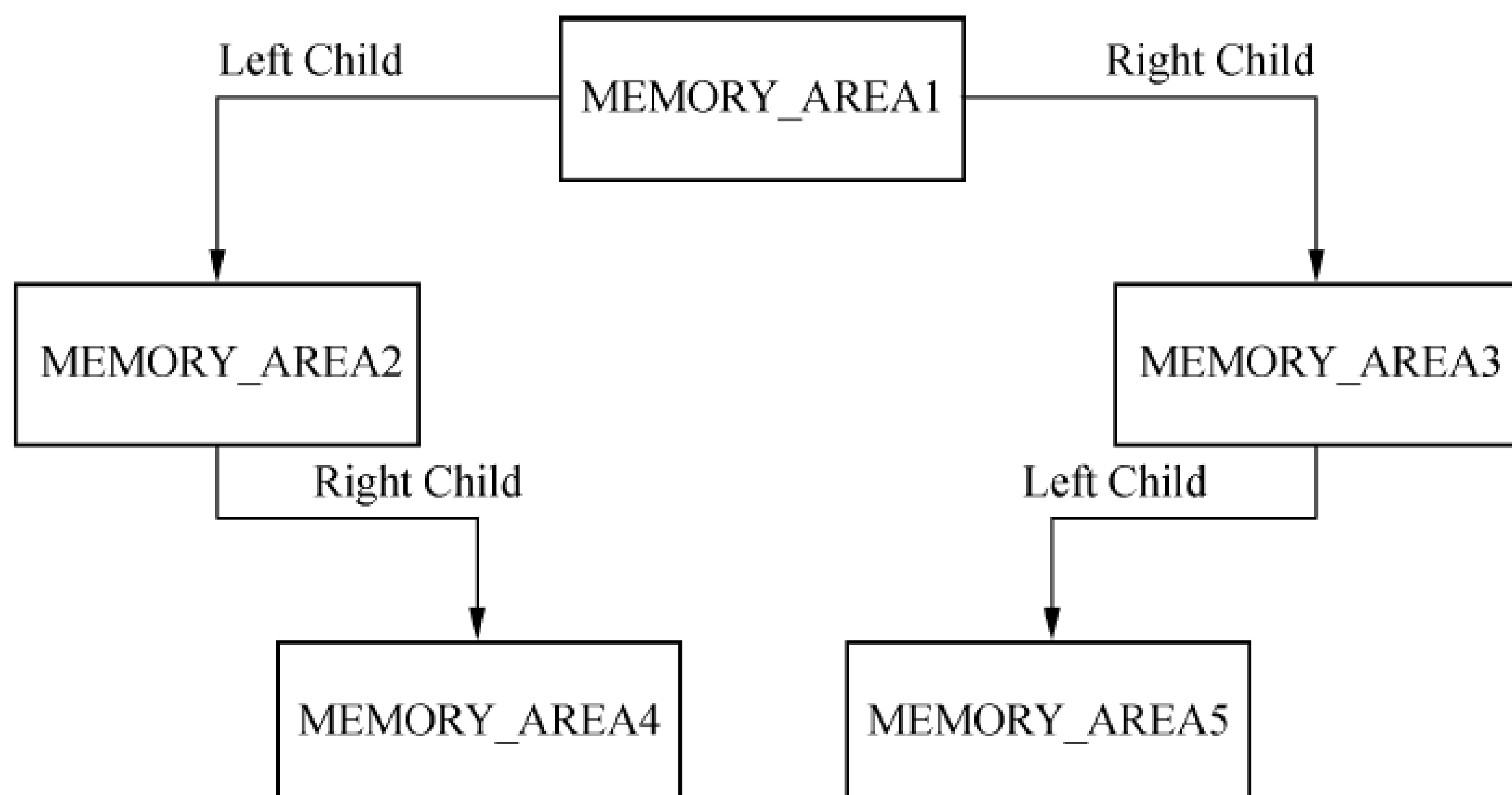


图 10 - 7　MEMORY_AREA 二叉树

2. 内存区块

每个区间 MEMORY_AREA 的 Data 域要么指向 SectionData 数据结构,要么指向区块队列 RegionListHead,如果 MEMORY_AREA 描述的区域是文件映射区或共享映射区,则 Data 域指向前者;如果只是普通内存区间,则指向后者。

区块队列 RegionListHead 中的元素是 MM_REGION 结构,一个 MM_REGION 是地址连续且属性和保护模式一致的地址范围(例如都是 MEM_COMMIT 或 MEM_RESERVE 类型,或都是 PAGE_READONLY 或 PAGE_READWRITE 保护模式)。这也从侧面说明相邻的 MM_REGION 必然属性不同或保护模式不同,否则就合并成一个 MM_REGION 了。

3. 内存页面

一个区块 MM_REGION 包含了若干虚存页面,因此区块 MM_REGION 是 4 KB 对齐的。页面是内存管理的最小单位。页面分为虚存页面和物理(内存)页面,后面还会讲述倒换页面,在 32 位系统中它们都是 4 KB 大小的,以方便以整页为单位进行换入换出操作。

由此,我们可以看出用户态虚拟内存空间的管理层次,由大到小分别是:内存空间→内存区间→内存区块→内存页面。接下来我们看看物理地址空间。

10.2.2　物理地址空间

相比虚拟地址空间,物理地址空间要简单很多,对于物理地址空间的管理同样也是以页面为单位进行的,但也只是页面这一种管理单位,不存在区块、区间和空间这些概念。在 Windows 系统中代表物理页面的数据结构是 PHYSICAL_PAGE,它有以下三种类型:





➢ **MM_PHYSICAL_PAGE_FREE**:表示已经被释放的物理内存页面。

➢ **MM_PHYSICAL_PAGE_USED**:表示正在使用中的物理内存页面。

➢ **MM_PHYSICAL_PAGE_BIOS**:表示 BIOS 专用的物理内存页面,其物理地址小于 0x100000。

在 Windows 内核中有一个内核初始化时创建的全局物理页面数组 MmPageArray,其下标就是物理页帧号(Page Frame Number,PFN),物理页面的地址右移 12 位(除以 4 KB)就是 PFN。系统中有多少物理内存页面,这个数组就有多少元素(4 GB 物理内存有 1 M 个页面,MmPageArray 数组的大小也为 1 M,页帧号的范围是 0 ~ 0XFFFFF)。MmPageArray 数组中的元素是 PHYSICAL_PAGE 结构,以下是该结构的具体定义:

```
typedef struct_PHYSICAL_PAGE
{
  ULONG Flags;                        //页标识,可以是上述三种类型之一
  LIST_ENTRY ListEntry;               //链表节点,用于挂入 7 个链表
  ULONG ReferenceCount;               //引用计数
  KEVENT Event;
  SWAPENTRY SavedSwapEntry;           //用于表示倒换页面所在的文件的页面列表
  ULONG LockCount;                    //锁计数
  Struct_MM_RMAP_ENTRY * RmapListHead; //本物理页面映射的那些虚拟页面组成的链表
} PHYSICAL_PAGE, * PPHYSICAL_PAGE
```

除了在 MmPageArray 数组中存放 PHYSICAL_PAGE,PHYSICAL_PAGE 结构也可以被挂入 7 个队列: BiosPageListHead、FreeZeroPageListHead、FreeUnZeroPageListHead 和 UsedPageListHeads 队列数组,其中 UsedPageListHeads 数组内含 4 个队列元素,如下所示:

➢ **MC_CACHE**:该队列用于内容缓存页面;

➢ **MC_USER**:该队列用于用户态空间映射的页面;

➢ **MC_PPOOL**:该队列用于可倒换到外存页面池的页面;

➢ **MC_NPPOOL**:该队列用于非分页内存页面池的页面。

由定义也可以看出 BiosPageListHead 是专门用于 BIOS 物理内存页面的队列,UsedPageListHeads 是用于在使用的普通物理页面的缓存队列,它与其他两个队列的关系如图 10-8 所示。

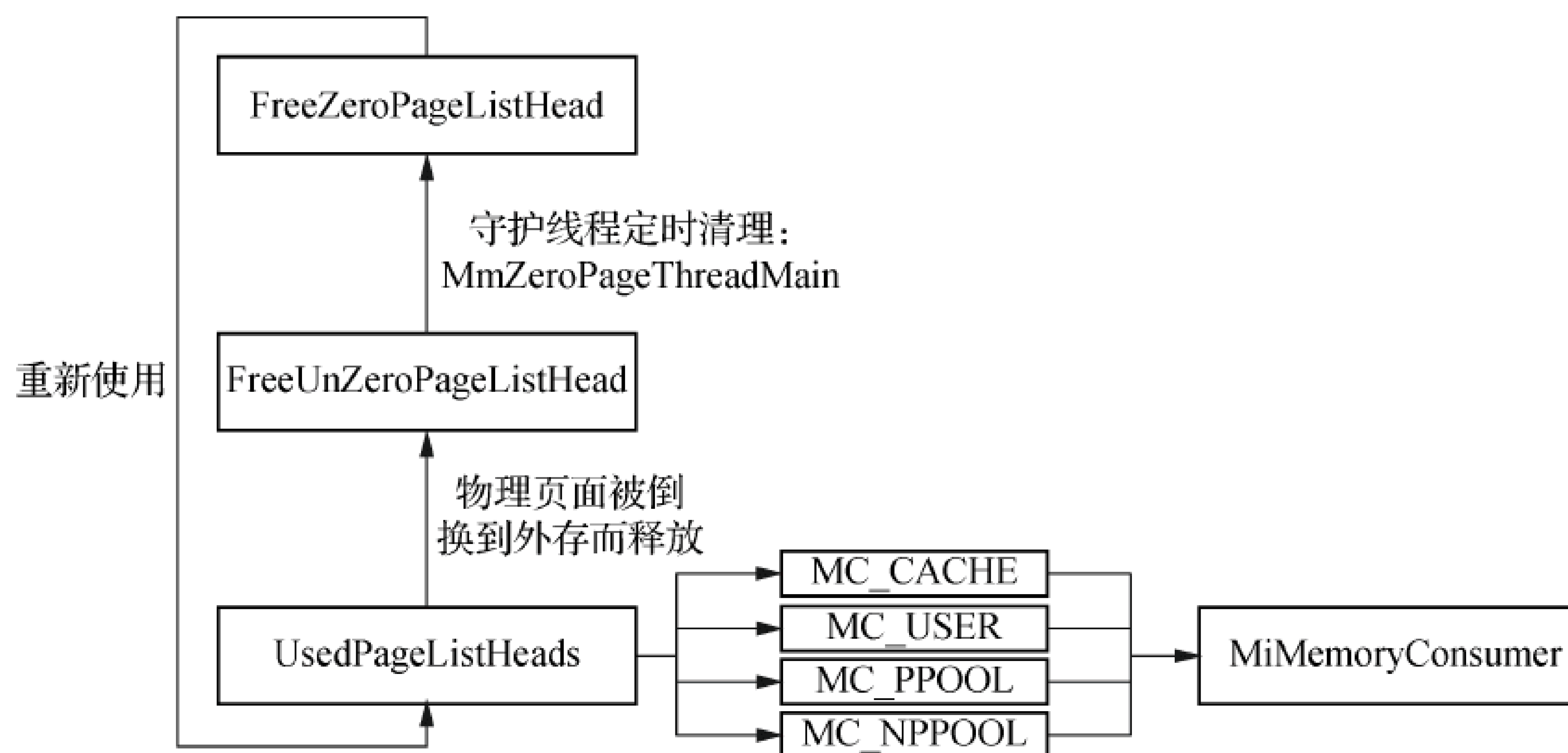


图 10-8　三个内存页面队列的关系

物理页面是周转使用的,而不是被分配后就一成不变的。与 UsedPageListHeads 相对应,内核中还有个数组 MiMemoryConsumer,与 UsedPageListHeads 一样也包含前述 4 个元素(MC_CACHE 等),该数组用于 4 种队列的页面配额,其元素是 MM_MEMORY_CONSUMER 数据结构,该结构包括了对已分配页面的描述、该队列的页面具体配额和修剪函数指针(每种队列类型的页面的老化修剪策略是不一样的,因此修剪函数也不一样,例如 MC_USER 队列的修剪函数是 MmTrimUserMemory)。

内存页面申请是采用 MmRequestPageMemoryConsumer 函数实现的。这个函数以队列类型为参数,对选定种类的队列进行配额计算和页面修剪,具体流程如图 10-9 所示。

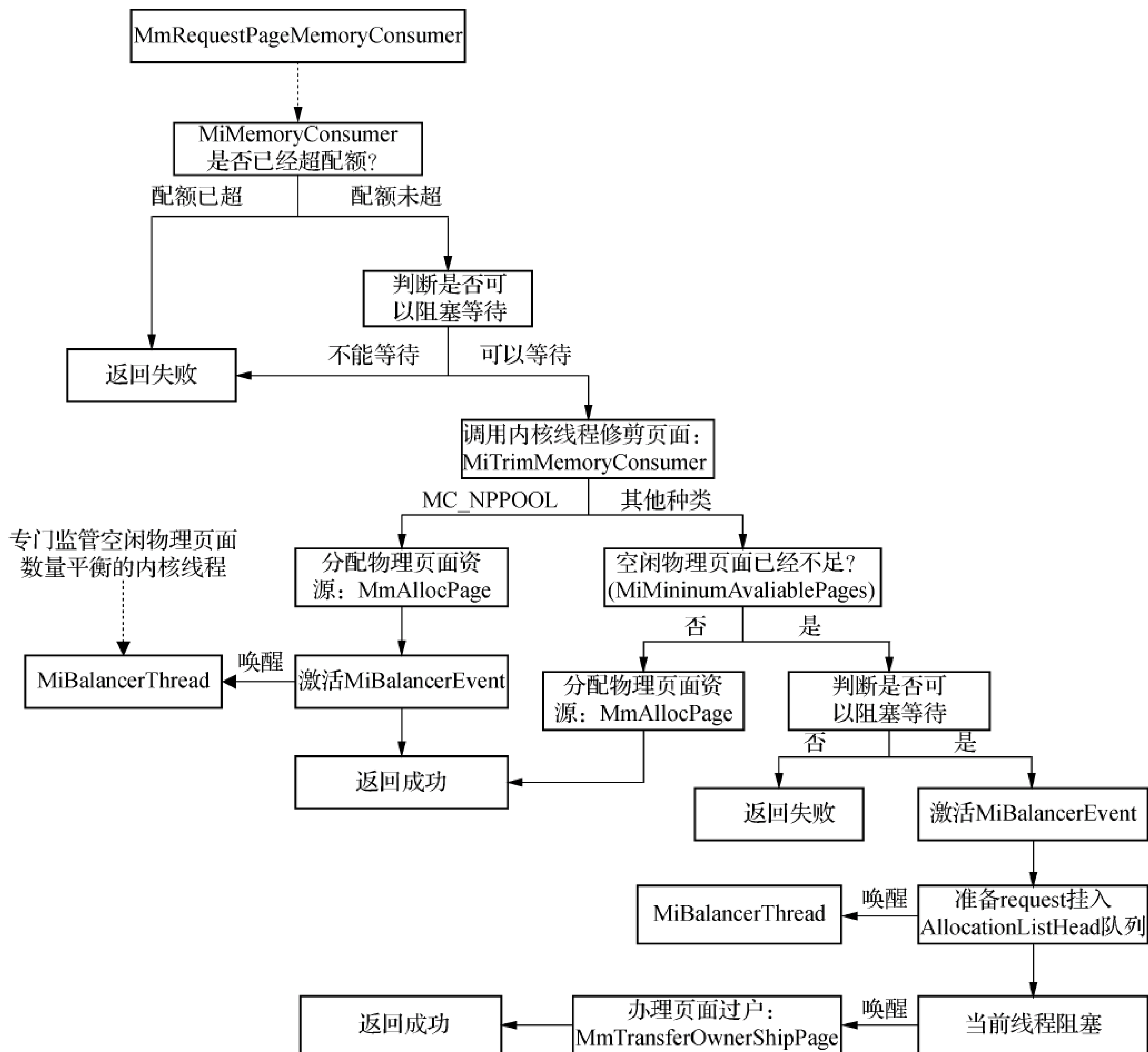


图 10-9　MmRequestPageMemoryConsumer 实现内存页面申请的流程

MmRequestPageMemoryConsumer 的主要流程逻辑是这样的:

(1) 首先判断对应请求的队列(例如 MC_CACHE)是否已经超配额了。这是通过判断内核队列数组 MiMemoryConsumer 实现的。

(2) 若已经超配额就不再分配物理页面了,此时选择返回失败。如果未超配额,要判断

该线程(内存页面申请线程)是否可以等待。因为页面的修剪和分配是个很耗时的过程,页面申请线程必须同步等待,不能等待就只能返回失败了,能等待则执行页面修剪函数。

(3) 如果可以等待,则调用内核函数 MiTrimMemoryConsumer 来修剪对应队列的物理页面。这个函数以所请求队列的类型为参数,调用 MiMemoryConsumer 数组中对应元素的修剪函数。

(4) 修剪完成后要区别对待以下两种情况:

- 对于 **MC_NPPOOL 类型的队列**,要特事特办,优先予以满足。因为非分页内存池的物理页面一般都是用在内核线程的,这是整个系统中优先级最高的线程,也理应优先予以满足。此时调用 MmAllocPage 函数分配物理页面,并激活 MiBalancerEvent 事件以唤醒 MiBalancerThread 内核线程。这个线程是专门监管空闲物理页面数量平衡的,如果内核页面需要通过修剪才能得到,这说明系统内的内存负荷已经很高了,需要对使用中的物理页面进行修剪释放。
- 对于**除 MC_NPPOOL 外的其他类型的队列**,判断其空闲的物理页面是否已经不足。如果还充足,则调用 MmAllocPage 分配一个物理页面,否则就要修剪一些物理页面了。此处修剪物理页面的步骤与 MC_NPPOOL 类型队列的不同,MC_NPPOOL 类型队列的优先级高,必须优先满足,无论如何都要挤出一个物理页面给它用。但其他队列的优先级没那么高,此时要启动修剪线程"现学现卖",这将是一个同步等待的过程,因此需要询问当前线程是否可以等待,不能等就直接 pass 了。

(5) 如果"现学现卖"过程中线程能等待,则准备一个物理页面申请 request,将其挂入全局队列 AllocationListHead 中,同时唤醒 MiBalancerThread 内核线程进行页面修剪。做完这些事情,当前线程就阻塞了(在 request 中的 Event 事件上等待)。

(6) 如果 MiBalancerThread 不孚众望地修剪出一堆空闲物理页面,AllocationListHead 队列中的请求应该被满足,这时要激活 request 中的 Event 事件,让被阻塞的页面申请的线程恢复执行。

(7) 对于分配过来的页面,由于它刚刚还是在其他进程使用的,现在"投诚"过来,所以要办理"过户手续":调用 MmTransferOwnerShipPage。至此,对物理页面的申请算是完成了。

页面分配函数 MmAllocPage 首先对 FreeZeroPageListHead 队列进行筛选,如果有空闲页面就直接返回;否则,再对 FreeUnZeroPageListHead 队列进行筛选。但 FreeUnZeroPageListHead 队列中的页面是要进行清洗的,比如零化页面,清洗完成后直接返回。

与 MmRequestPageMemoryConsumer 相对应,MmReleasePageMemoryConsumer 是用于页面释放的内核函数,这个函数就非常简单了:从 AllocationListHead 队列中获取下一个页面申请(request),然后把即将释放的页面挂入该 request 的 page 域并激活 Event 事件,这样在该 request 上等待的线程就被唤醒了。

10.2.3　页面映射机制

1．页面映射

所谓页面映射，是指虚拟内存和物理内存发生关联，即根据页面目录表通过虚拟内存找到物理内存的具体页面，或者倒换文件的具体页面。只有映射了物理内存的虚拟内存才可以被 CPU 访问。

页面映射通过内核函数 MmCreateVirtualMapping 实现，具体流程如下：

（1）将物理页面标记为“已映射”，使用内核函数 MmMarkPageMapped 实现。

（2）去掉该物理页面原来的映射，使用内核函数 MmMarkPageUnMapped 实现。

（3）将物理页帧的地址左移 12 位并与页面属性相并后赋值给 PTE。物理页面的大小为 4 KB，因此一个物理页帧的地址（页帧号）必定是 4 KB 对齐的，也就是页帧号的低 12 位是无效的，只有高 20 位的描述才有意义。但是 PTE 是 4 字节的，也就是说存放了页帧号后还有 12 位是空闲的，因此可以用这 12 位存放一些页面属性。

（4）递增该页面的页面映射表的引用计数。

（5）刷新页面映射表项在 TLB 中的映像，通过内核函数 MiFlushTlb 实现。

页面映射的删除通过内核函数 MmDeleteVirtualMapping 实现，具体步骤如下：

（1）使用内核函数 MmGetPageTableForProcess 获取当前页面的 PTE。

（2）将页面目录表中该页面的 PTE 置零。

（3）刷新页面映射表项在 TLB 中的映像，通过 MiFlushTlb 函数实现的。

（4）去掉该物理页面原来的映射，通过 MmMarkPageUnMapped 函数实现。

（5）释放物理页帧，通过 MmReleasePageMemoryConsumer 函数实现。

（6）递减该页面的页面映射表的引用计数。

2．页面倒换与倒换页面

在现代操作系统中，不但有虚拟内存页面和物理内存页面，还存在一种倒换页面。倒换页面本质上是磁盘上的页面，是系统中用于文件后备的页面。在操作系统中引入倒换页面主要是为了节省内存，例如在系统运行大量进程时会需要占用较多内存资源，将当前暂时不被访问的内存上的内容写回磁盘文件（如果内存页面未被更改则不需要写回磁盘文件）从而释放物理内存页面以供其他进程使用；或者在系统休眠时将长时间未被访问的物理页面（其更改过的“脏”物理页面写回磁盘文件）收回。可使用内核函数 MmPageOutVirtualMemory 将虚存页面的内容倒出到倒换页面中，具体步骤如下：

（1）断开页面目录表对应的 PTE 与物理页面之间原有的联系；

（2）断开物理页面与倒换页面之间原有的联系；

（3）建立 PTE 与倒换页面之间新的联系；

（4）清零 PTE 的物理页标志位 PA_PRESENT，使 PTE 与物理内存页面之间的映射失效。





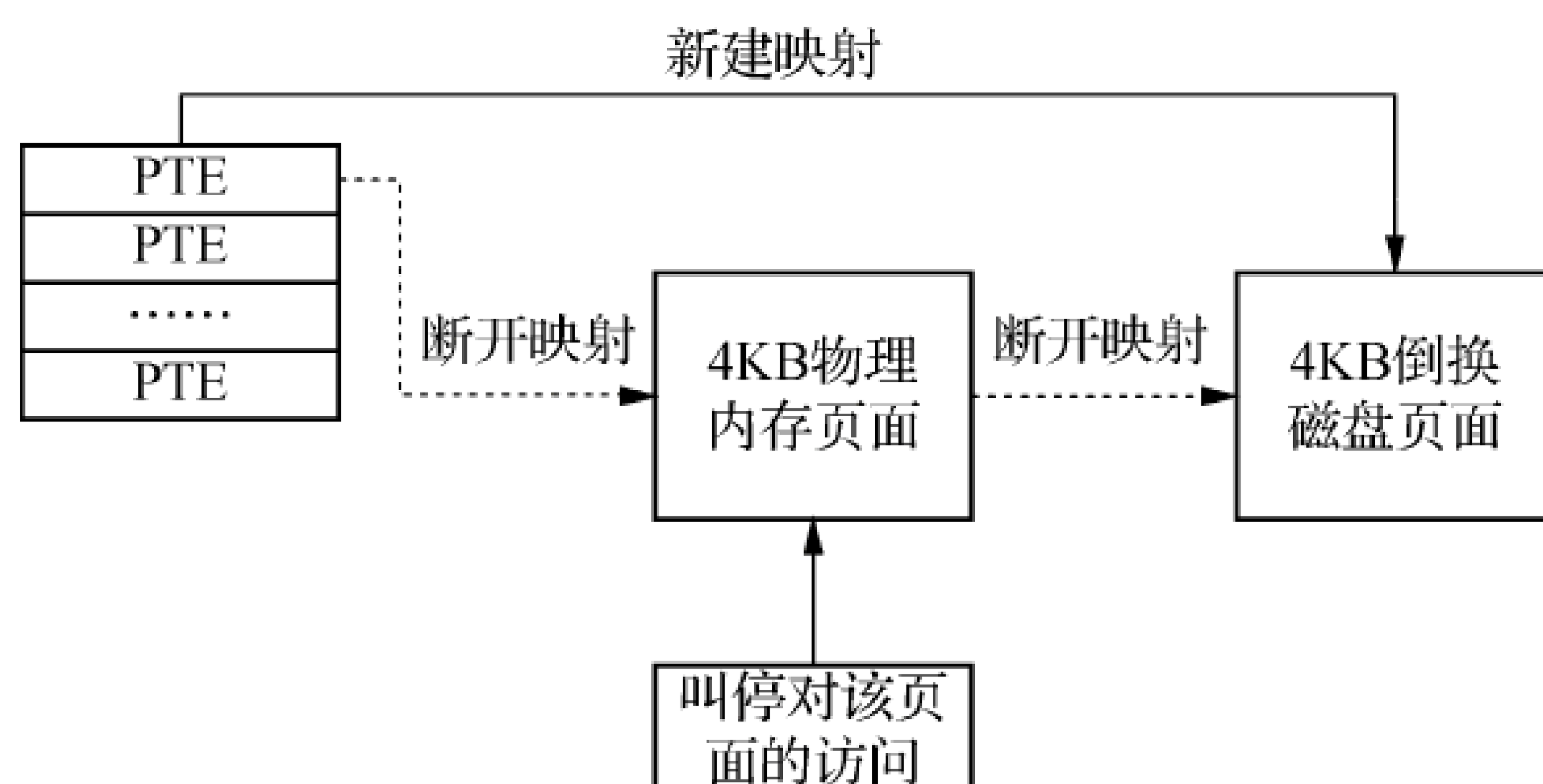


图 10-10　倒换文件页面映射示意图

可以看出,一个虚存页面要么与物理页面映射,要么与倒换页面映射,要么无映射(对应的 PTE 为空)。当 PTE 的 PA_PRESENT 标志位为 0 时表示页面不在内存中,CPU 访问到该页会产生缺页中断,中断例程将该页面对应的后备文件倒换到内存中来。当一个页面长时间未受到访问就会被倒出到倒换页面中。倒入不是系统的自发行为,而是缺页中断发生后的被动行为,因此倒入的过程应包含在缺页中断响应函数(MmAccessFault)中。

我们先来看看页面倒出函数 MmPageOutVirtualMemory 的主要执行流程,如图 10-11 所示。

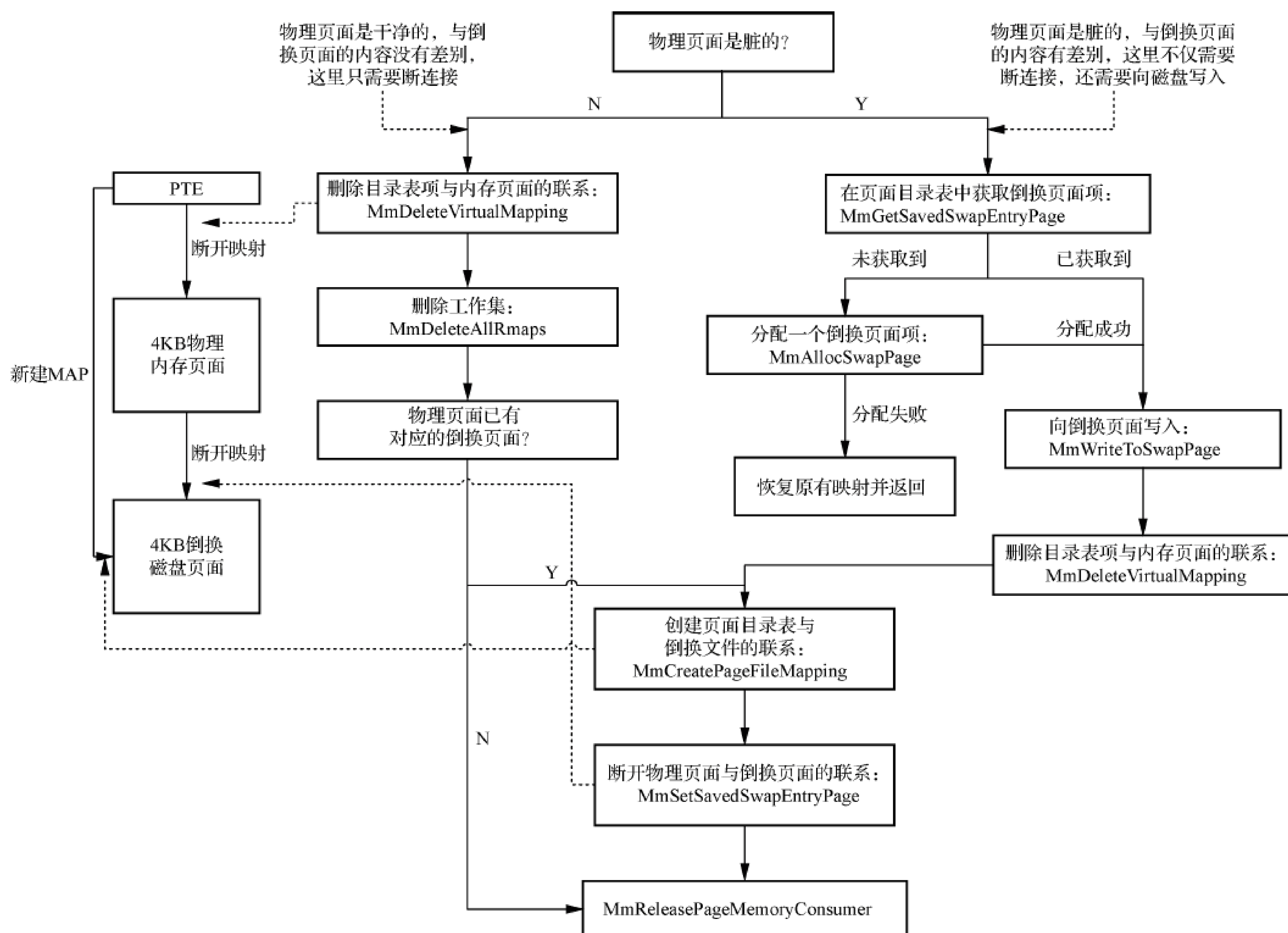


图 10-11　MmPageOutVirtualMemory 的主要执行流程

所谓“工作集”，就是当前进程保留在物理内存中的页面，这些页面或者正在被访问，或者刚被访问还未倒换到后备文件，或者短期内还可能再被访问。

MmPageOutVirtualMemory 首先判断当前的物理页面是否是“脏”页面，所谓脏，就是在本次被调入的过程中页面中的内容发生了改变，需要写回倒换文件中保存。这很好理解。基于此我们分为干净页面和脏页面两种情况讨论。

先看干净页面的处理过程：

（1）首先删除页面目录表项与物理内存页面的映射关系。

（2）删除工作集（物理页面不再可用了，因此需要把工作集里面这个物理内存页面删掉以维护数据一致性）。

（3）检查这个物理页面是否有对应的倒换页面：

- 如果有，则创建页面目录表与倒换文件之间的映射（以备下次再用），删除物理页面与倒换页面之间的映射，最后释放物理页面。
- 如果没有，则直接释放该物理页面。

而针对脏页面的处理过程如下：

（1）首先在页面目录表中获取倒换文件页面的目录表项（页面目录表有两种目录表项：物理页面项和倒换页面项），如果未能获取到目录表项，则分配一个目录表项。

（2）有了目录表项（刚刚新建或已有）后将当前物理页面的“脏”数据写回倒换文件。

（3）删除对应的目录表项与物理内存页面的映射关系。

（4）创建页面目录表与倒换文件之间的映射（以备下次再用），删除物理页面与倒换页面之间的映射，最后释放物理页面。

此时 PTE 与物理页面、倒换页面的关系如图 10－12 所示。

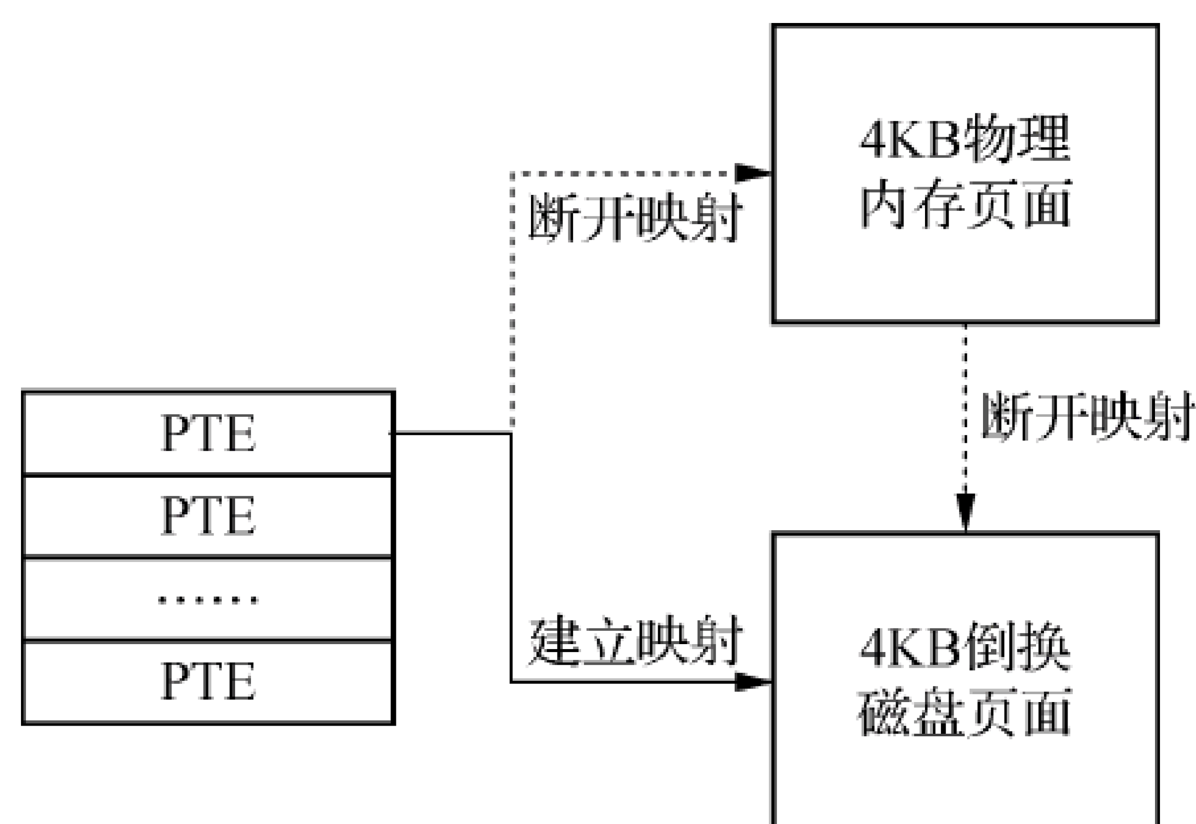


图 10－12　建立 PTE 与倒换页面的映射

以上我们说的是进程的用户态地址空间。但是一个进程不光有用户态空间，还有内核态空间，仍然要靠页面目录表来映射。可以想象，在 32 位系统的进程页面目录表中，PDE 表的后半部分代表了 2 GB 的内核态空间，这部分内容在各个进程中都基本一致，因此可以通过内核函数 MmCopyMmInfo 从系统内核映射表 MmGlobalKernelPageDirectory 中复制内核态空间部分的映射（只需要更改两处不一致的地方就行了）。





我们说基本一致，但其实内核部分有两处不一致，即每个进程的页面目录表和超空间。进程页面目录表作为4 GB空间中的一部分，其固定的虚拟内存位置是0xC0000000，也就是说每个进程从0xC0000000开始的4 MB虚拟地址空间是不一样的。同样，超空间是起点地址为0xC0400000的4 MB虚拟地址空间，以备临时周转时使用，如图10－13所示。

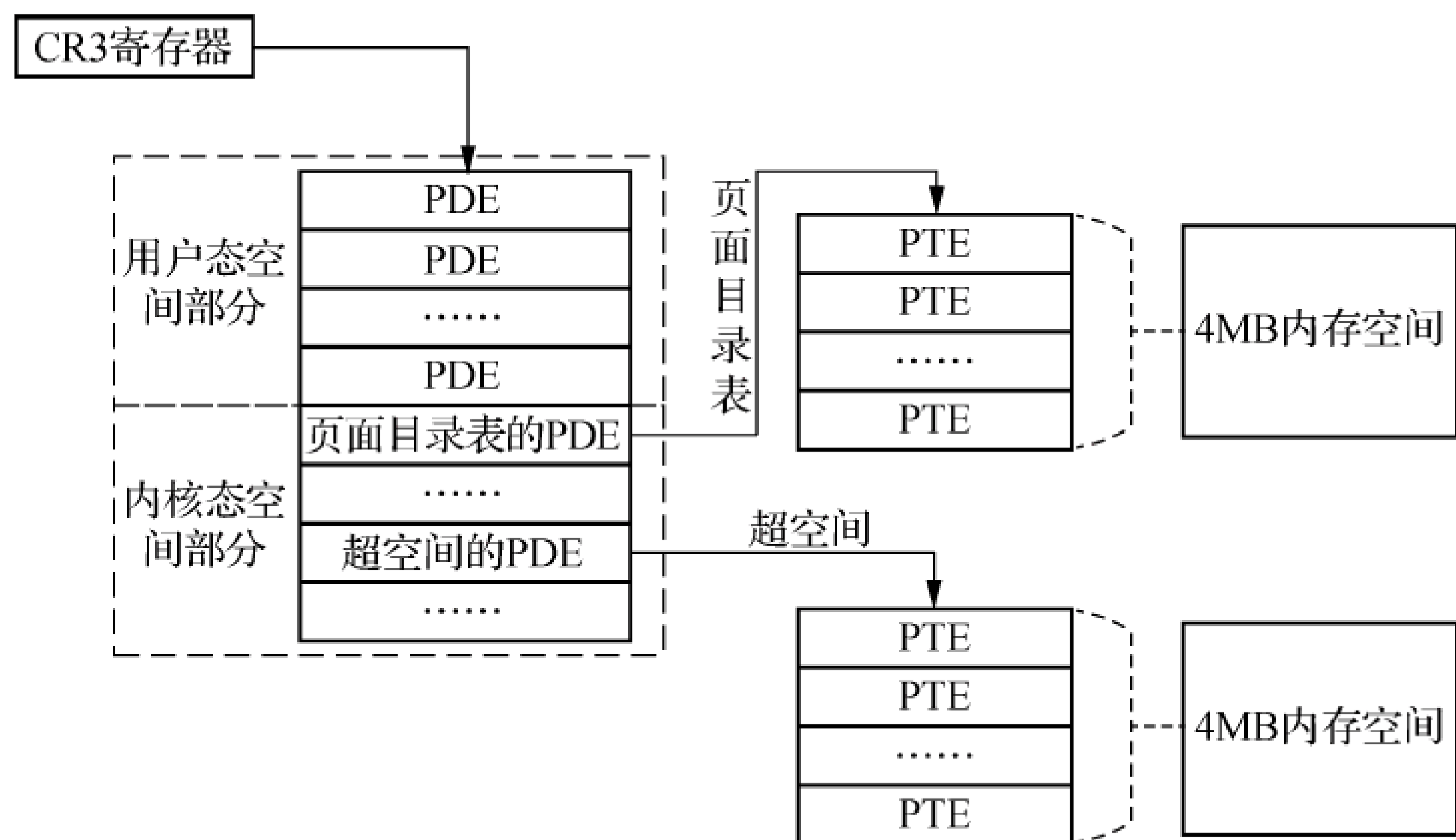


图10－13　三级页面目录表示意图

创建进程时通过MmCopyMmInfo从系统内核映射表中复制内核部分的映射，并且调用MmCreateProcessAddressSpace创建进程的地址空间，可见目录表在创建进程时就要构造好。

3．内存分配

下面我们来看系统函数NtAllocateVirtualMemory，这个函数的作用是分配内存空间并修改相关区域的属性。前文说过，内存空间的管理粒度从大到小为空间→区间→区块→页面，NtAllocateVirtualMemory正是实现了这个管理链条，只不过最后的页面映射处理过程要与异常处理机制配合完成。NtAllocateVirtualMemory的执行流程如图10－14所示。

NtAllocateVirtualMemory首先判断是否指定了起始地址BaseAddress：

- **指定了BaseAddress**：通过内核函数MmLocateMemoryAreaByAddress找到BaseAddress所在的区间（MEMORY_AREA），由于该区间已经存在，并且可能需要修改该区域页面的属性，例如可读、可执行、提交保留状态等，因此要考虑该区域所辖区块的修改。如图10－14中左半部分所示，BaseAddress原本对应的区块是区块2，由于BaseAddress所在页面属性的修改，区块2要分裂成3块，前后两块依然保留原来的属性，中间一块的属性被NtAllocateVirtualMemory传进来的参数修改。
- **未指定BaseAddress**：不指定地址就无法从内存空间中找到这个内存区间，只能通过内核函数MmCreateMemoryArea创建，并初始化区间的区块列表RegionListHead。

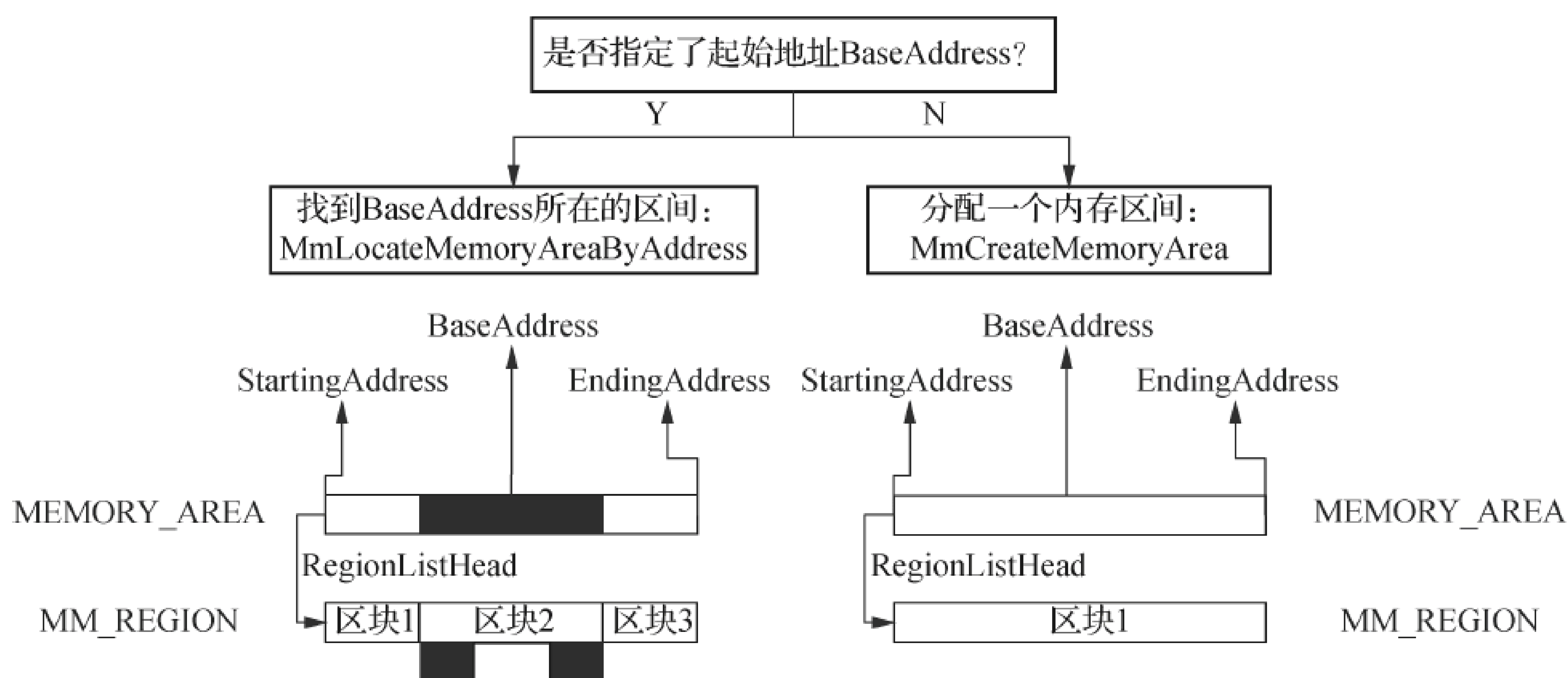


图 10-14　NtAllocateVirtualMemory 的执行流程

注意：BaseAddress 只是 MEMORY_AREA 中间的一个点，不应狭义地看成 StartingAddress，因为 BaseAddress 不一定是 4 KB 对齐的，而内存区间 MEMORY_AREA 包含了区块，区块又包含了页面，因此 MEMORY_AREA 的 StartingAddress 一定是 4 KB 对齐的。

10.2.4　页面换出机制

页面从后备文件中倒入物理内存，这是个被动的行为，即当缺页异常发生时，如果要访问的内容在后备文件中，要将该页面的内容倒入物理页面中。但是页面的倒出却是个主动行为，在 Windows 中有个内核线程 MiBalancerThread 负责将进程正在使用的物理页面内容倒换到后备文件以便释放一些物理页面。

MiBalancerThread 等待两类事件：

- **定时器事件 MiBalancerTimer**：由定时器激活；
- **平衡器事件 MiBalancerEvent**：由事件激活。

激活后线程 MiBalancerThread 被唤醒，它的工作很简单，就是调用 MiMemoryConsumer 数组中每个元素的修剪函数，以释放这个队列中某些老化的物理页面。元素就是物理页面的消费者，系统中有四大页面消费者：用户态空间进程（MC_USER）、内核分页池（MC_PPOOL）、内核非分页池（MC_NPPOOL）和文件缓存（MC_CACHE）。一般都是从用户态空间进程和文件缓存两个消费者的页面队列中“压榨”物理页面。例如 MC_USER 元素代表的页面队列的“压榨”（修剪）函数就是 MmTrimUserMemory。

MmTrimUserMemory 的逻辑也很简单，即采用“最近最少被用到”的 LRU 算法将符合条件的物理页面倒换到后备文件中（由 MmPageOutPhysicalAddress 实现）。倒换出去的页面分为两类：

- **共享映射区页面**：采用 MmPageOutSectionView 函数将页面倒换出去；
- **普通映射区页面**：采用 MmPageOutVirtualMemory 函数将页面倒换出去。

倒换的步骤大家已经了解了，这里不再赘述，如图 10-15 所示。

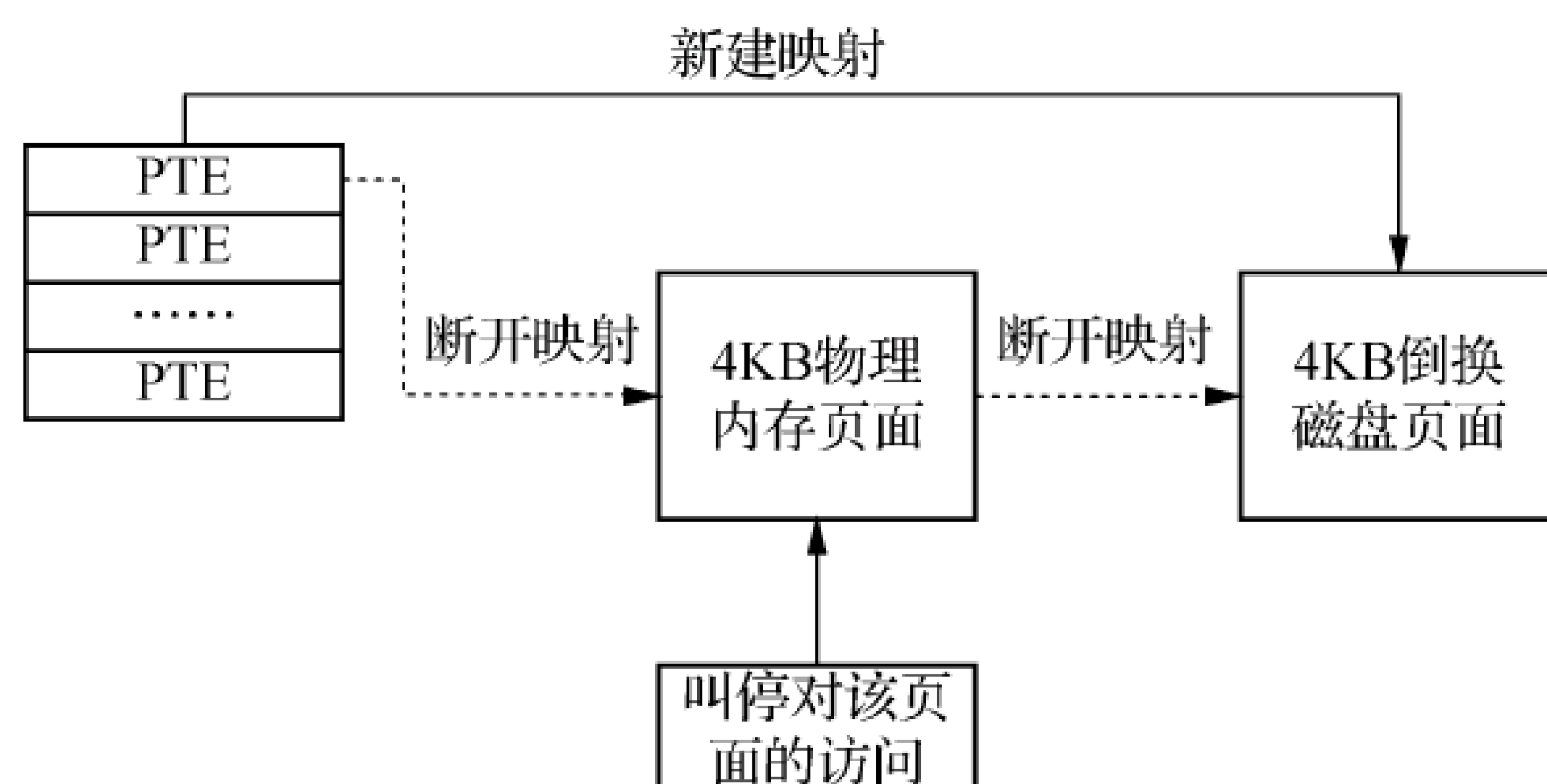


图 10-15　页面换出过程

10.3　内存页面异常处理

在第9章中介绍异常处理时曾经描述过,当出现内存页面异常时,CPU 会从 IDT 查找并执行对应的异常处理例程。内存页面异常的处理例程是 KiTrap14。内存页面异常分为两类:缺页异常和访问越权。缺页异常是常态化异常,因而并不是真正意义上的异常,这也是本节要介绍的内容;而访问越权就是真正意义上的异常了,例如在不可执行的内存页面执行代码。访问越权是包括堆栈溢出漏洞在内的系统安全问题发生的根源。

处理缺页异常的内核函数是 MmAccessFault,其具体执行流程如下:

(1) MmAccessFault 首先判断缺页异常发生的地址是在内核态空间还是用户态空间,如果在内核态空间,则调用 Mmi386MakeKernelPageTableGlobal 更新当前进程的系统空间映射。因为发生在这个空间区域的缺页异常很可能是由于系统的内核空间映射发生了变化而进程的内核态空间未及时更新造成的。

(2) 如果缺页异常发生在用户态空间就要区分到底是越权访问造成的异常还是真正的缺页异常:

- 如果是越权访问异常,则调用 MmpAccessFault 处理;
- 如果是缺页异常则调用内核函数 MmNotPresentFault 处理。

如图 10-16 所示是 MmNotPresentFault 的处理流程:

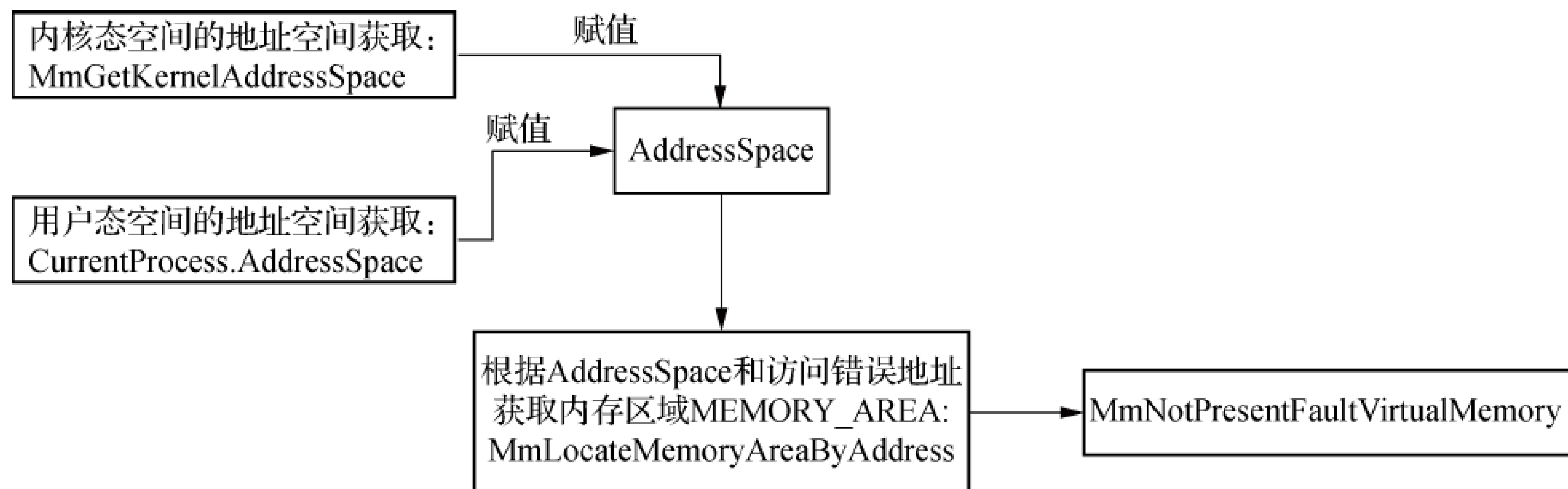


图 10-16　MmNotPresentFault 的执行流程

MmNotPresentFault 根据发生异常的地址是在用户态空间还是内核态空间分别获取对应的 MADDRESS_SPACE（实参为 AddressSpace），并以此实参调用 MmLocateMemoryAreaByAddress 以获取内存区间 MemoryArea，最后调用真正的处理函数 MmNotPresentFaultVirtualMemory 来处理页面异常。

页面异常的实质处理函数 MmNotPresentFaultVirtualMemory 的处理流程如下：

（1）获取出错地址所在的内存区块 Region，判断该 Region 是不是保留状态或者不允许访问的状态，是则出错返回，不是则执行下一步。

（2）创建或获取一个页面申请操作。

- 之所以说“创建或获取”，是因为一个物理页面往往对应了好几个虚存页面，也就是说可能有好几个线程会对这个物理页面进行访问，可能别的线程已经产生了该页的缺页异常，针对这个物理页面的申请操作已经存在了。这种情况下只需要静静阻塞等待申请操作完成就好，只要完成了，无论是谁，大家都可以访问，不需要重复创建申请操作。
- 当阻塞的线程继续执行时，说明已经获取到了物理页面，函数返回就可以了。
- 当然如果之前没有其他线程发起针对此页面的申请，那就由当前线程来发起，做“第一个吃螃蟹的人”：执行 MmRequestPageMemoryConsumer 以分配内存页面。这个函数执行两次，第一次不等待，如果第一次不成功则再执行一次，这一次要等待（漫长的页面修剪操作）。如果还是分配不成功就直接返回失败了。

（3）分配物理页面成功后，判断是否存在倒换文件页面。

① 如果存在，则：

- 清零页面映射表中对应 PTE；
- 使用内核函数 MmReadFromSwapPage 从倒换文件中读取页面内容；
- 使用内核函数 MmCreateVirtualMapping 建立映射关系；
- 使用内核函数 MmInsertRmap 将物理页面插入工作集；
- 激活步骤（2）中创建或获取的页面申请操作的等待事件。

② 如果不存在，则只执行上述最后三步。

10.4　共享映射区机制

一个物理页面可以只属于一个进程，也可以属于多个进程，对于属于多个进程的情况我们称为共享映射区机制。物理页面以磁盘文件作为倒换文件页面，利用共享映射区机制可以做到共享多个进程的文件内容，如图 10－17 所示。

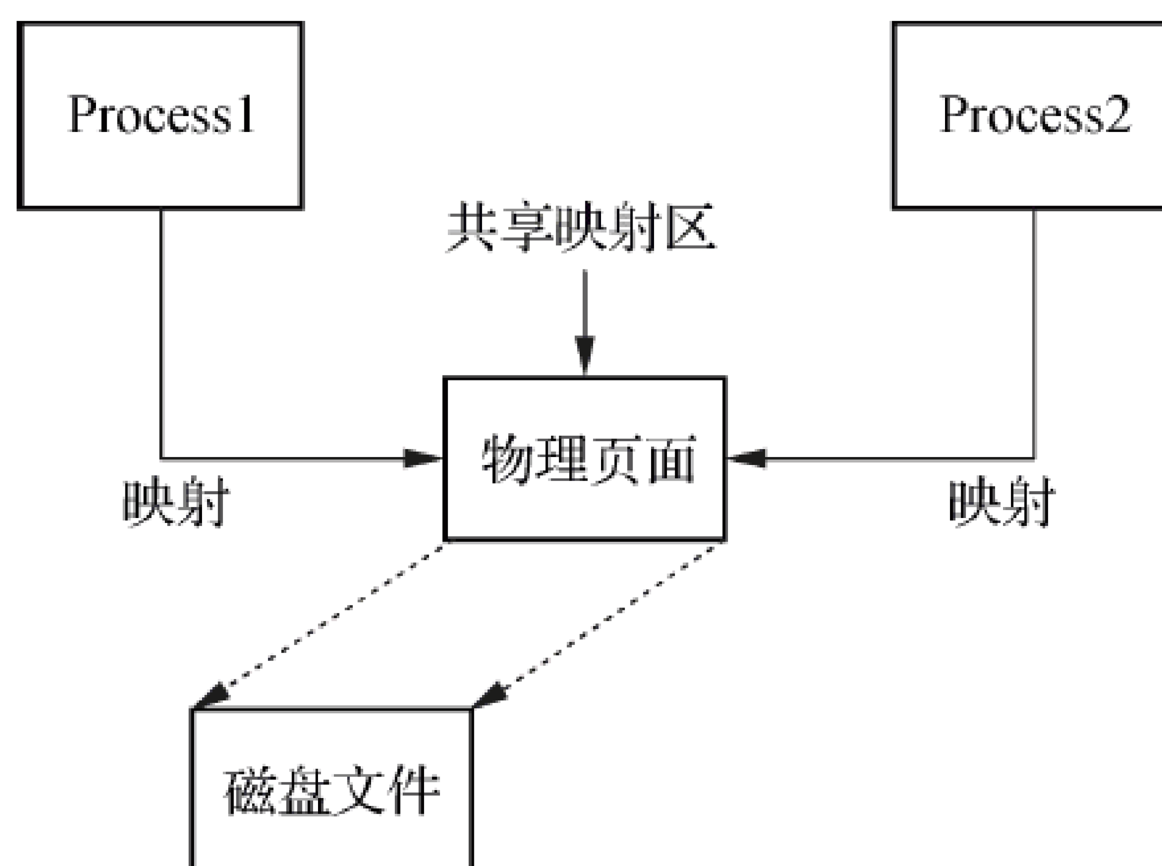


图 10－17　共享映射区机制

系统调用 NtCreateSection 用于创建内存映射

区,按映射文件种类的不同,NtCreateSection 要分别调用以下几个子函数,这几个子函数也是 NtCreateSection 的逻辑主体:

- 对于 Windows PE 文件的共享映射区调用 MmCreateImageSection。PE 文件包含了数据段、代码段、资源段等多个区段,因此对于 PE 文件的映射要考虑多个区段的问题。
- 对于普通文件映像的共享映射区调用 MmCreateDataFileSection。普通文件映像只有一个数据段,因此普通文件映像可以看作 PE 文件映像的一个特例。
- 对于无目标文件的共享映射区(物理页面)则调用 MmCreatePageFileSection。创建多进程之间的共享映射区时采用这种方式。

我们以最简单的 MmCreateDataFileSection 为例来讲解创建共享映射区的过程:

(1) 创建映射区对象 Section,这是由对象创建函数 ObCreateObject 实现的,实参为 MmSectionObjectType,表明这是一个映射区对象类型。

(2) 找到需要映射的文件对象,即通过对象查找函数 ObReferenceObjectByHandle 并以文件句柄为参数查找文件对象 FileObject。

(3) 通过 I/O 函数 IoQueryFileInformation 探查文件长度,以确定映射区大小的上限。

(4) 如果目标文件尚未映射(FileObject 的映射区对象指针中尚未建立区段),则为其创建一个映射数据段 Segment,这个数据段的类型为 MM_SECTION_SEGMENT。

如图 10-18 所示,Section 作为映射区对象,一方面关联了需要映射的文件对象,另一方面要建立若干区段,例如对于 PE 文件要建立与代码段、数据段、资源段等相对应的 Segment,这时的 Section 类型为 IMAGE_SECTION_OBJECT,而 Section 采用 ImageSegments 指针来指向这些 Segment;对于普通文件映像,Section 类型为 SECTION_SEGMENT。Segment 比较重要的字段有 FileOffset 和 PageDirectory。FileOffset 表示该区段映射的内容在被映射文件内部的偏移(可能文件不会被全部映射,只是映射其中的一部分)。PageDirectory 是个指向映射数据段页面表的指针,映射数据段页面表与页面目录表基本一致,也是个二级目录表,只不过映射数据段页面表以目标文件中的页面号为下标,如图 10-19 所示。

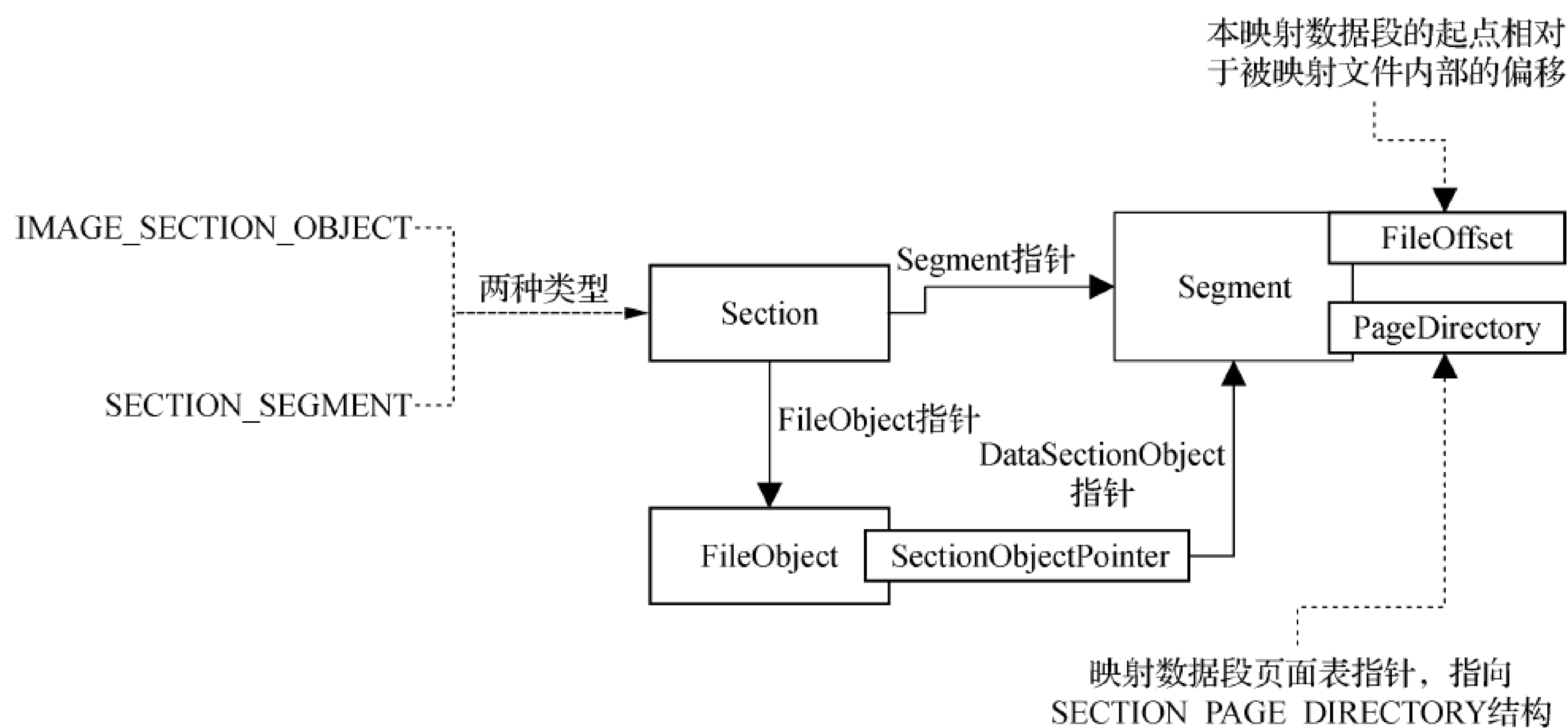


图 10-18 Section 与 Segment 结构示意图

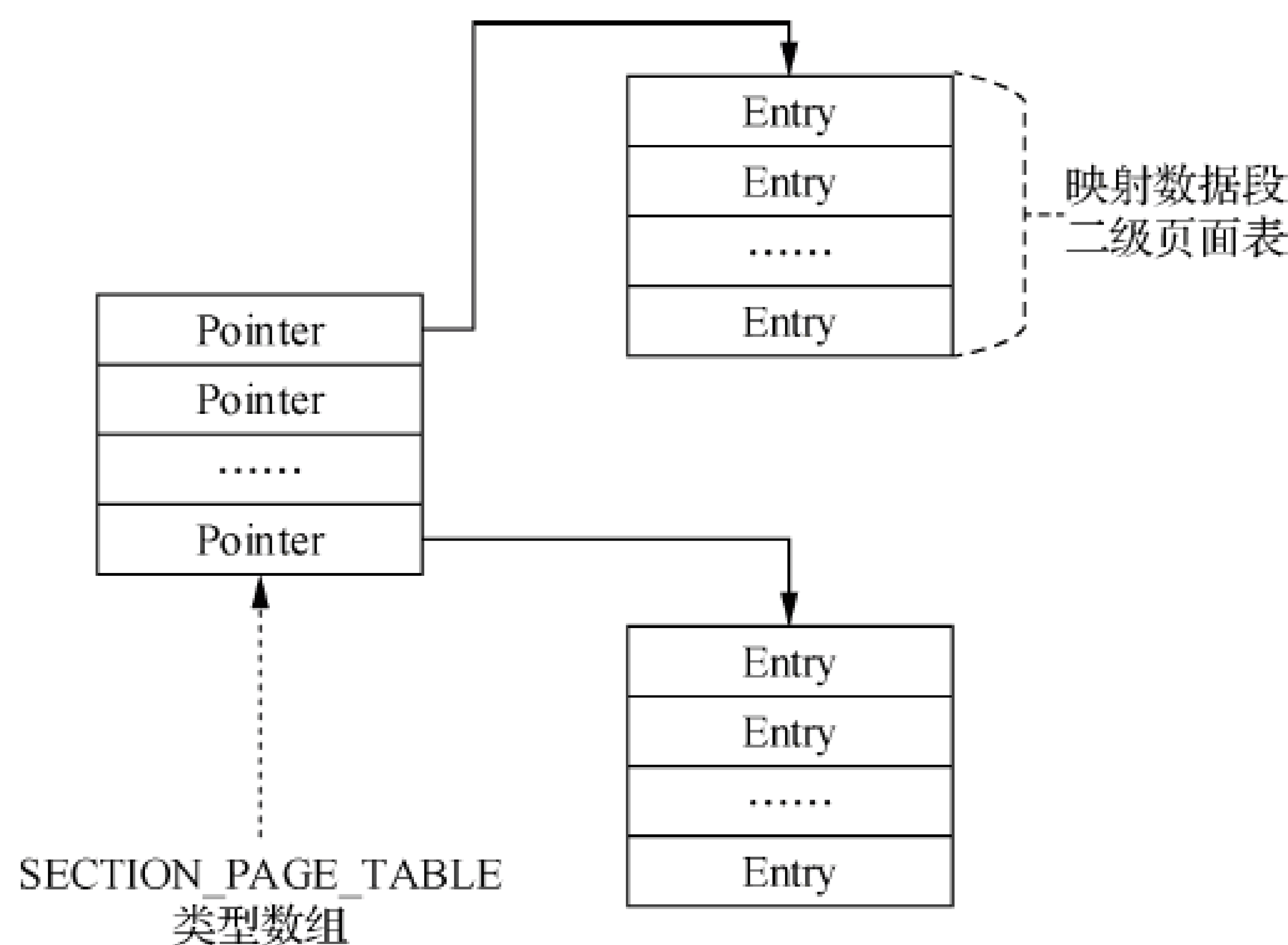


图 10-19　映射数据段页面表结构示意图

一级和二级页面表的每个表项都是 4 字节，而每个表又都有 1 K 个表项，因此与页面目录表一样每一个映射数据段页面表都占用一个 4 KB 页面。只不过二级页面表的下标是页面在目标文件中的页面号（目标文件就相当于整个虚拟地址空间）。

MmCreateDataFileSection 经历了上述千辛万苦，也只是创建了诸如 Section 和 Segment 这样的数据结构，要想使用映射区还要创建映射关系，这与之前讲的页面映射机制的道理是一致的。

负责创建映射关系的是系统调用 NtMapViewOfSection，该函数将一个映射区对象的一部分或全部映射到某个进程的用户态地址空间中。NtMapViewOfSection 的核心是 MmMapViewOfSection，而这个函数又调用 MmMapViewOfSegment 来映射区段（例如对于 PE 文件要映射好几个区段）。MmMapViewOfSegment 的核心操作是通过 MmCreateMemoryArea 在进程的地址空间中创建这部分的内存区间，这又回到我们熟悉的函数了。

MEMORY_AREA 中的 Data 域是个联合类型，当这个内存区间是虚拟内存类型时指向 VirtualMemoryData 数据结构，这已在前文有所描述；当为映射区类型时指向 SectionData 数据结构。在本节中自然就是后者了，这样串联下来，根据虚拟地址可以找到进程的内存区间，从而找到 Section 乃至找到 Segment，如图 10-20 所示。

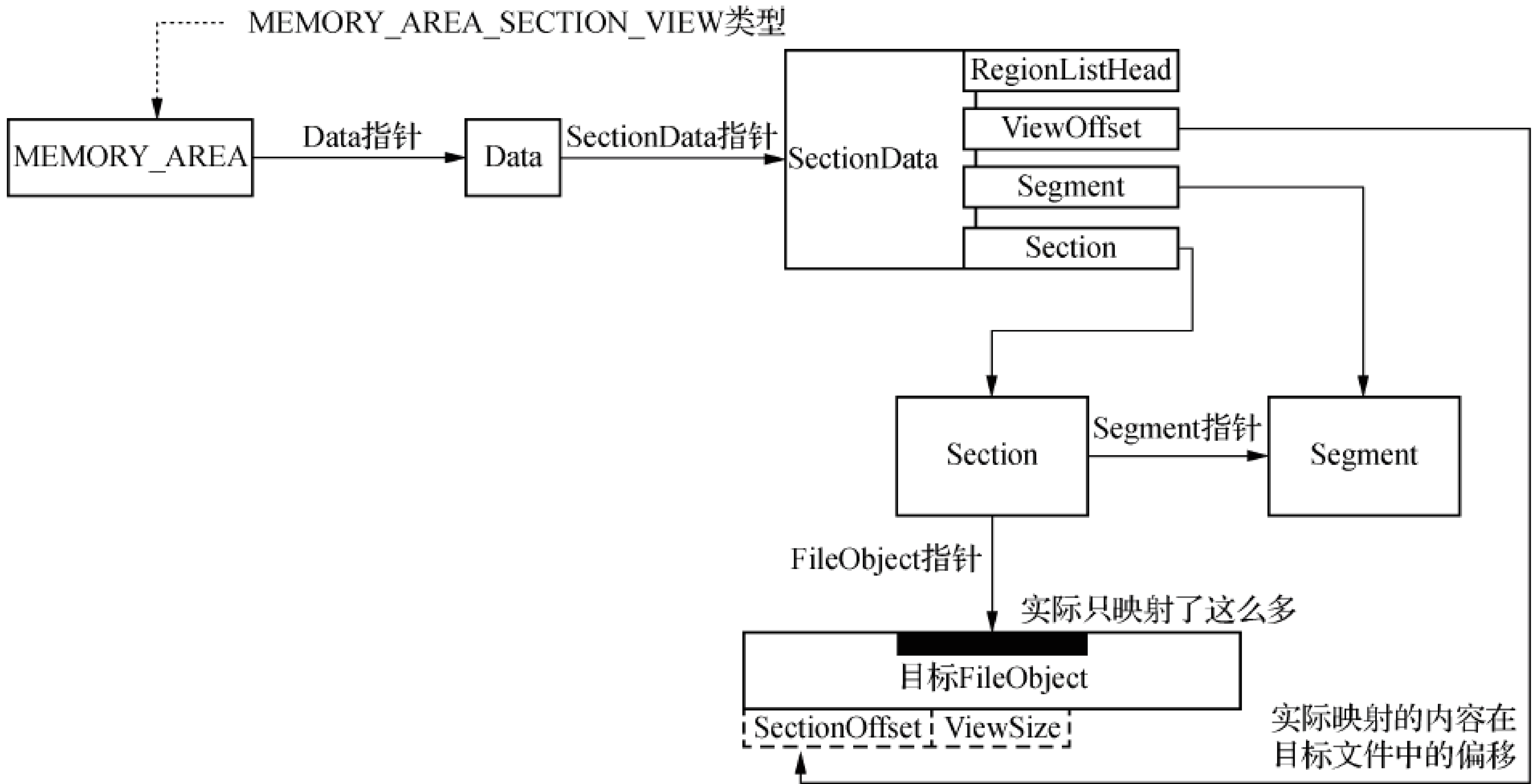


图 10-20　内存区间、内存区段与文件对象的关系

从图 10－20 看出，从 MEMORY_AREA 出发可以找到 Segment，而在 SectionData 中也保存了映射区在文件中的偏移等重要信息。

10.5 内存管理机制的改进

从 Windows Vista 开始，在本章介绍的内存管理机制的基础上，Windows 对内存管理又做了重大改进，包括：

➢ 大量使用无锁同步技术或者更细粒度的锁，以提高多核架构下的内存管理性能；
➢ 支持更大的换页 I/O 吞吐；
➢ 支持 GPU 内存结构管理；
➢ 更有效地使用 TLB 硬件；
➢ 使用闪存作为磁盘与内存之间的中间缓存（Ready Boost 技术，利用了闪存随机读写及零碎文件读写上的优势来提高系统性能）；
➢ 支持动态内核地址空间划分机制，以解决地址范围紧缺的问题（目前还只支持 32 位系统，64 位系统仍然采用静态划分方式）；
➢ 支持 SuperFetch 机制，利用内存可用空间预加载用户可能使用的应用程序页面，尽可能地避免系统发生同步的硬盘页面调用，让应用程序以最快的速度运行。

不过 SuperFetch 也是一项颇具争议的技术。例如在 Windows 系统中，如果在一定时间内没有界面操作，类似杀毒软件这样的高内存消耗进程就会被系统启动，而原先用户运行的应用进程的物理内存页面可能被换出。等到再次进行界面操作时，杀毒软件等被系统自动启动的进程的页面会被换出，原先运行的应用进程被换入。这个反应的周期特别长，给人的感觉是系统很“卡”。

而 SuperFetch 技术会跟踪页面的使用历史，例如跟踪当前页面的使用信息和曾经出现在内存中的代码与数据信息，以便指示内存管理器将大概率能用到的磁盘文件中的数据和代码预加载到备用链表中，以避免发生那种“书到用时方恨少”的现象。但是这种技术也是把双刃剑，其有效的程度取决于学习的准确性和有效性，如果准确性较低，会加载很多无用的页面，反而拖累了系统效率。

10.6 64 位系统下的内存空间

与 X86 体系结构相比，X64 体系结构下表示的地址范围更大，因为名义上是 64 位的地址线。

但实际上 X64 CPU 对于地址线的使用做了限制，只使用低 48 位，高 16 位闲置不用。因此 X64 体系结构下的地址空间划分是这样的：

➢ **用户态空间地址范围**：0x00000000`00000000 ~ 0x0000FFFF`FFFFFFFF；
➢ **内核态空间地址范围**：0xFFFF0000`00000000 ~ 0xFFFFFFFF`FFFFFFFF。

可见地址空间的中间有一大片地址范围是不使用的。而 Windows 64 位系统对地址的使用又做了更进一步的限制,即只使用低 44 位,高 20 位闲置不用,因此其地址空间就被划分成了两块 8TB 的连续空间,如下所示:

- **用户态空间地址范围**:0x00000000`00000000～0x000007FF`FFFFFFFF;
- **内核态空间地址范围**:0xFFFFF800`00000000～0xFFFFFFFF`FFFFFFFF。

在 X64 体系结构下,地址空间的布局也与 X86 体系结构下的差异很大。图 10－21 是 X64 体系结构下的内核态空间布局。

Start	End	Size	Description
FFFF0800`00000000	FFFFF67F`FFFFFFFF	238TB	Unused System Space
FFFFF680`00000000	FFFFF6FF`FFFFFFFF	512GB	PTE Space
FFFFF700`00000000	FFFFF77F`FFFFFFFF	512GB	HyperSpace
FFFFF780`00000000	FFFFF780`00000FFF	4K	Shared System Page
FFFFF780`00001000	FFFFF7FF`FFFFFFFF	512GB-4K	System Cache Working Set
FFFFF800`00000000	FFFFF87F`FFFFFFFF	512GB	Initial Loader Mappings
FFFFF880`00000000	FFFFF89F`FFFFFFFF	128GB	Sys PTEs
FFFFF8a0`00000000	FFFFF8bF`FFFFFFFF	128GB	Paged Pool Area
FFFFF900`00000000	FFFFF97F`FFFFFFFF	512GB	Session Space
FFFFF980`00000000	FFFFFa70`FFFFFFFF	1TB	Dynamic Kernel VA Space
FFFFFa80`00000000	*nt!MmNonPagedPoolStart-1	6TB Max	PFN Database
*nt!MmNonPagedPoolStart	*nt!MmNonPagedPoolEnd	512GB Max	Non-Paged Pool
FFFFFFFF`FFc00000	FFFFFFFF`FFFFFFFF	4MB	HAL and Loader Mappings

图 10－21　X64 体系下内核态空间布局

在 64 位 Windows 系统下,上图中某些地址段超过了 44 位地址线的分布范围,但也只是在 Windows 内部使用,并不作为通用的存储空间。

最后要注意的是,由于 64 位系统下 PDE/PTE 的长度为 8 字节,因此一个内存页面(4 KB)中只能包括 512 个 PDE/PTE。对于 PDE 来讲,一个 PDE 可对应 512 个 PTE,一个 PTE 仍然代表了一个 4 KB 的页面,因此一个 PDE 只能表示 2 MB 的内存空间,这也是虚拟地址空间的默认分配粒度。

本章小结

本章介绍了在 Windows 中内存的管理,这里的内存包括虚拟内存和物理内存。介绍了虚拟地址空间和物理地址空间的概念、页面映射和换出机制、页面缺页异常处理过程、共享映射区机制等。最后介绍了 64 位系统下的内存空间。

第11章　进程间通信机制

进程间通信(Inter-Process Communication,IPC)是操作系统的重要机制,包含了多种实现手段,例如本地过程调用、远程过程调用、命名管道、邮件槽、信号量、等待唤醒、互斥锁、事件、共享内存区、TCP/IP、DCOM、WMI 等。这么多种手段,面向的也是不同的应用场景,包括以下两大类:

- **进程间数据报文交换**:本地过程调用、远程过程调用、命名管道、邮件槽、共享内存区(见前文)、消息队列、TCP/IP、DCOM、WMI 等;
- **线程间协同**:信号量、等待唤醒、互斥锁、事件等。

鉴于关于线程间协同的公开资料非常多,这里不再赘述,本章重点介绍进程间数据报文交换中的本地过程调用和命名管道,同时也会介绍跨主机进程间的交互方式 WMI(虽然一般不把 WMI 算作 IPC 手段),这几种也是目前较为"Well Known"的方式。

本章将按照图 11－1 所示的提纲对这三者进行介绍。至于其他手段,TCP/IP 机制和消息队列作为不同主机进程间通信的基本手段将在后文介绍;远程过程调用是基于 TCP/IP 机制的,因此理解了 TCP/IP 机制,远程过程调用问题就迎刃而解了;邮件槽机制目前用得非常少,这几种手段就不占用篇幅了。

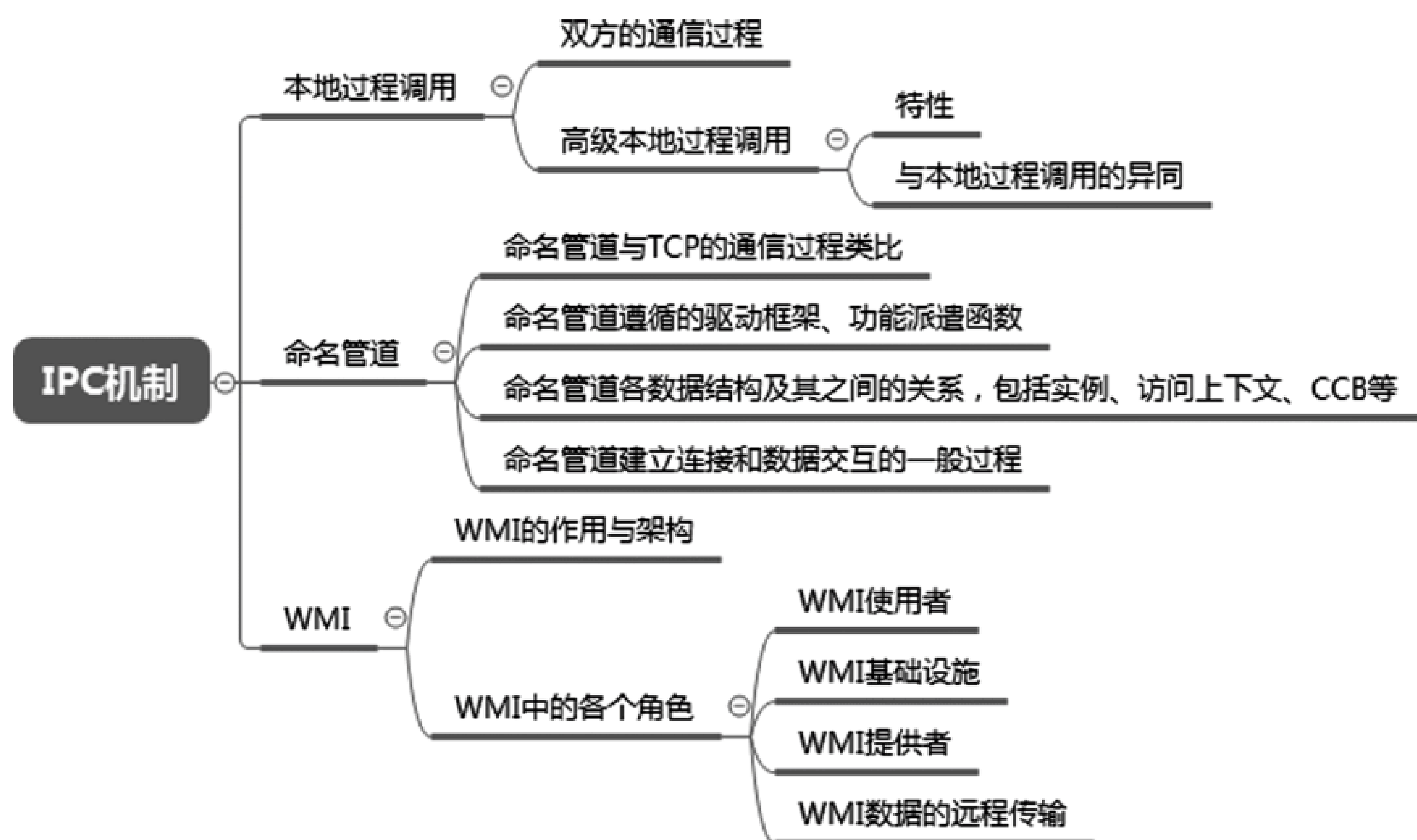


图 11－1　本章提纲

11.1　本地过程调用

11.1.1　本地过程调用通信过程

本地过程调用(Local Procedure Call,LPC)是同一主机操作系统进程间通信的重要手段。本地过程调用与远程过程调用(Remote Procedure Call,RPC)是相对的。LPC 是发生在同一主机内的通信行为,故而称为“本地”;而 RPC 既可以用于同一主机内通信,也可以用于不同主机间通信,故而称为“远程”。例如创建进程线程时通知环境子系统进程 csrss.exe 就是采用 LPC 机制。

RPC 一般基于 TCP/IP 协议进行通信,其将服务封装为 API 供应用进程调用,调用后请求转化为 RPC 报文并通过 TCP/IP 协议在主机间交互。LPC 不采用 TCP/IP 协议,而是采用一种被称为“端口通信”的机制,虽然这种方式的运作过程与 TCP 方式的通信非常相似,但其实是风马牛不相及的两回事。

这里要强调一下,虽然说是“进程间通信”,但实际上通信的双方是线程(线程才是操作系统调度的基本单位),因此 LPC 本质上是本地系统中两个线程间的数据拷贝和操作同步。在线程执行体数据结构 ETHREAD 中,凡是带 LPC 的域都是与本地过程调用相关的,例如:

- LpcReplyChain:用于将本线程挂入目标端口的待应答线程队列;
- LpcReplyMessage:指向本线程发出的请求报文,并用来回收应答报文;
- LpcReceivedMessageId:本线程最近收到的尚未应答的报文的序号;
- LpcReplyMessageId:已发送的请求报文的序号,其应答的序号也应该相同。

LPC 与之前描述过的视窗型报文有些像,两者都是通过线程间同步的手段来操作信息的交互,而数据的拷贝又都是通过共享映射区机制实现的。

我们说 LPC 本质上是端口机制。端口分为通信端口和连接端口两类,前者负责报文传输、数据交互,后者负责线程间建立连接。LPC 将线程也分为服务端线程和客户端线程,特别像 TCP 通信中的服务端与客户端的概念:

- 客户端线程向服务端线程建立连接以建立通信通道;
- 服务端线程监听端口以等待客户端线程的连接;
- 服务端线程在收到客户端线程建立连接的请求后接受该请求;
- 客户端线程与服务端线程建立连接后,双方各维持一个端口,称为通信端口。

从上述流程可以看出,其交互服务的过程与 TCP 方式的网络通信服务的过程非常相似,而 LPC 依赖的端口机制也是借用了 TCP 的端口概念(例如连接端口类似于 TCP 监听端口,通信端口类似于已经建立连接后的传输端口)。由于流程非常简单(如图 11 - 2 所示),这里就不再详细介绍了,但要注意以下几点:

- 第一步中创建的命名端口,其名称需要让客户端线程知道,否则就无法打开端口建立连接。
- TCP 与 LPC 都是双工的,即客户端与服务端都可以主动发送数据。

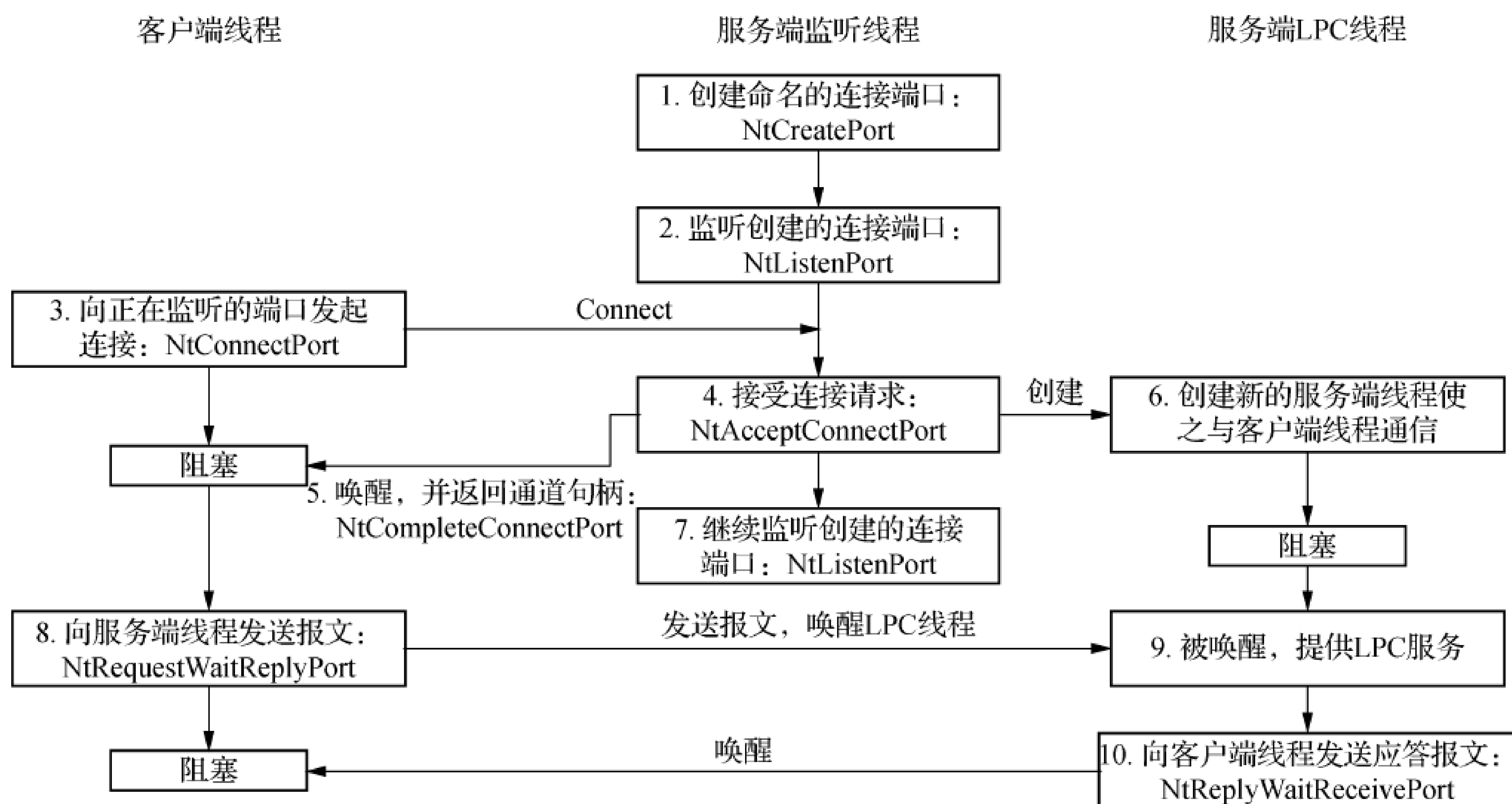


图 11－2　本地过程调用的通信流程

➢ 所谓报文发送，就是将报文挂到对方端口的报文队列中，并将对方线程唤醒。
➢ 报文传输过程中，如果客户端发送的报文＋数据的总长度不大于 256 字节，则数据随着报文一起发送，即在同一个缓冲区中传输；如果大于 256 字节，就要通过共享内存来发送，具体地说，客户端线程和服务端线程同时映射一块共享内存，通过 LPC 进行同步控制，通知数据到达，如图 11－3 所示。

客户端线程
报文传递
服务端线程
ClientView：映射
ServerView：映射

图 11－3　大数据量的 LPC 传输

11.1.2　高级本地过程调用

不过，随着高级本地过程调用（Advanced Local Procedure Call，ALPC）的引入，传统的 LPC 机制趋于淡化。ALPC 是一种高速、可伸缩、安全的 IPC 机制，可以传递任意大小的消息，因此被应用于与 csrss. exe 通信、Winlogon 与本地安全认证子系统服务进程 lsass. exe 通信、错误报告等诸多场景。

ALPC 与 LPC 的通信机制相同，都具有连接端口和通信端口，并且都是基于 TCP 式的监听和连接机制，消息传递的机制也大致相同，但 ALPC 增加了一个消息队列，这是消息传递过程中最大的不同。同时，ALPC 支持三种消息交换方式，且支持消息取消机制，如下所示：

➢ **双缓冲机制**：内核从发送线程中拷贝了要发送的消息，然后切换到目标线程，目标线程从内核拷贝这个消息。这种发送方式与传统 LPC 的发送方式大致相同。

➢ **ALPC 对象区机制**：发送端和目标方线程都映射该内存区的视图，这与传统 LPC 机制也并无不同。
➢ **消息区机制**：消息存放在内存描述符列表（MDL）构成的消息区中，这个消息区对应的物理页面被映射到内核态地址空间中，使得双方的线程都映射这一部分内核态内存空间，而这部分空间无论是虚拟地址还是物理地址都是固定的。这种机制的效率比双缓冲机制要高得多。

ALPC 也对异步通信机制做了改进，提供了三种不同的异步通信模型：

➢ **ALPC 完成列表机制**：ALPC 内部的完成列表是一种非阻塞的数据结构，使用者可以自由选择同步方法（被动事件机制或主动查询机制等）。
➢ **基于完成端口的机制**：完成端口是 Windows 中的一种 I/O 请求机制，后文会详细介绍，使用该机制可以使线程每次获取多个消息，提高 I/O 性能。
➢ **基于执行体回调对象的通知机制**：使用者向 ALPC 注册回调接口，当 ALPC 接收到消息时调用该接口，并将报文回调给注册方。

11.2　命名管道

命名管道（Named Pipe）作为 Windows 传统的 IPC 机制目前已经很少见了，这是因为更简单、兼容性更好、通用性更强、吞吐量更大的 IPC 机制（例如 TCP/IP）越来越成熟和多样化。但是作为 Windows 仍然保留的一部分功能，我们还是简要介绍一下其原理。

LPC 作为操作系统的固有机制是与内核绑定在一起的，涉及的数据结构（如 ETHREAD）和系统调用也都是内核固有的组件和接口。但命名管道不同，它是基于 Windows 的管道驱动实现的，与原生内核体系的运作机制关系不大，命名管道遵循设备驱动框架。

命名管道是双工通信的，既可以是单点对单点模式，也可以是多点对多点模式。同时命名管道支持跨主机方式通信，这比本地过程调用更具有普适性。命名管道的通信流程与本地过程调用非常相似，都遵循 TCP 式的“打开端口→监听端口→建立连接→接受连接→发送报文”的一般流程，如图 11－4 所示。

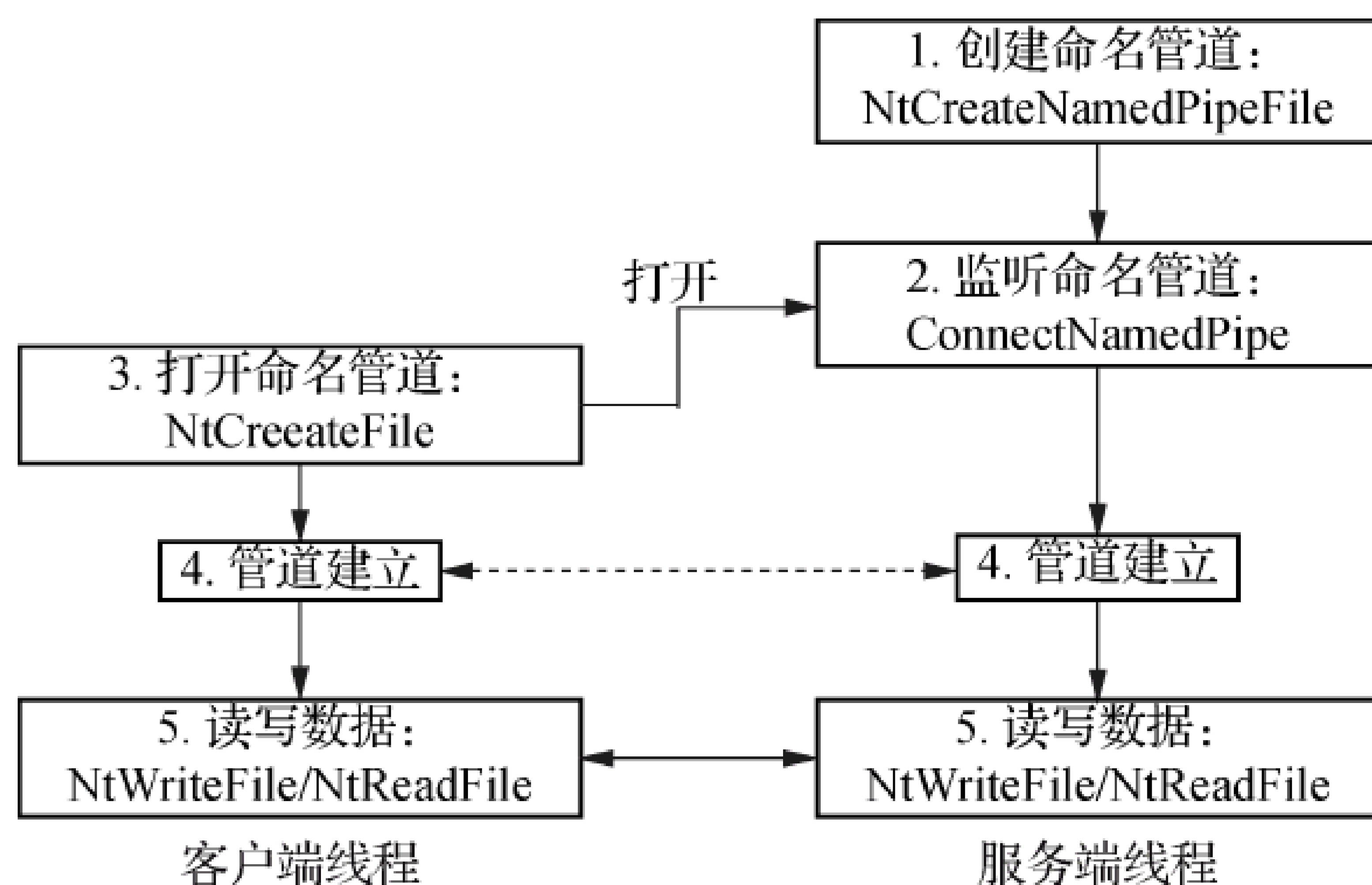


图 11－4　命名管道的通信流程

从图 11－4 可以看出，除了创建和监听命名管道之外，其他函数都是普通的系统调用。其实这些调用（包括创建和监听命名管道）本质上还是与驱动程序打交道，因为它们在系统调用序列的最后是与 I/O 管理器打交道的，I/O 管理器会将针对设备（命名管道在操作系统中被看作 I/O 设备）的系统请求分解成 IRP（I/O 请求包），进而与驱动程序通信。

例如管道创建函数 NtCreateNamedPipeFile 最终调用 I/O 管理器的 IoCreateFile，作为实参的设备类型为 CreateFileTypeNamedPipe，这是 Windows 命名管道的设备对象类型；而系统调用 NtCreateFile 用来打开命名管道设备，作为实参的设备类型则是更为普适的 CreateFileTypeNone。

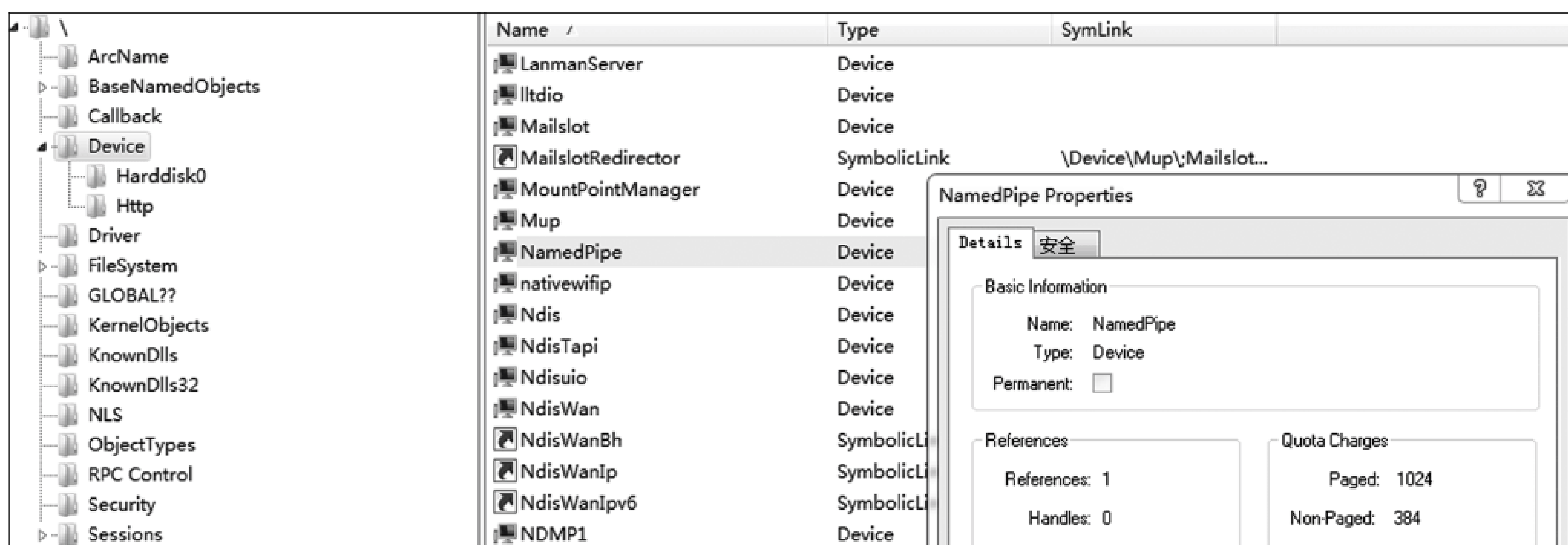


图 11－5　Windows 7 下的命名管道设备对象（尚未创建设备实例）

通过 IoCreateFile 创建了命名管道设备对象后，在整个系统的对象目录中就会出现对应的节点，例如“\Device\NamedPipe\XXX”，如图 11－5 所示。图 11－5 只是展示了命名管道设备对象，创建了管道设备实例后会在 NamedPipe 节点下生成所创建的设备实例名称。在上面这个目录中，设备对象 NamedPipe 提供对象解析函数，以方便设备管理器根据句柄或设备路径找到具体设备对象所属的目录，从而找到具体的设备实例。

前文说过，命名管道遵循的是设备驱动框架，即存在命名管道驱动模块，每个驱动模块都会有一个入口函数 DriverEntry，就像 DLL 模块也有个入口函数 DllMain 一样，在驱动程序安装的时候被调用。驱动的形态一般是 SYS 文件，与 DLL 文件一样，都是 Windows PE（可移植执行）文件，如图 11－6 所示。后面的章节中我们会对驱动框架和 PE 文件做详细介绍，这里并不赘述。



图 11－6　npfs. sys 的 PE 结构

在 Windows 中命名管道的驱动程序是 npfs. sys，其入口函数负责创建管道设备对象，并初始化设备对象的参数域值。驱动程序是在系统初始化的时候加载的，因此在用户登录系统的时候，驱动程序已经加载完了，命名管道设备对象也创建完毕了。

任何设备对象都分为对象本身和扩展区两部分，扩展区承载了诸多重要的属性，例如设备的功能派遣函数、与该类设备相关的特定属性等。命名管道设备对象如图 11－7 所示。

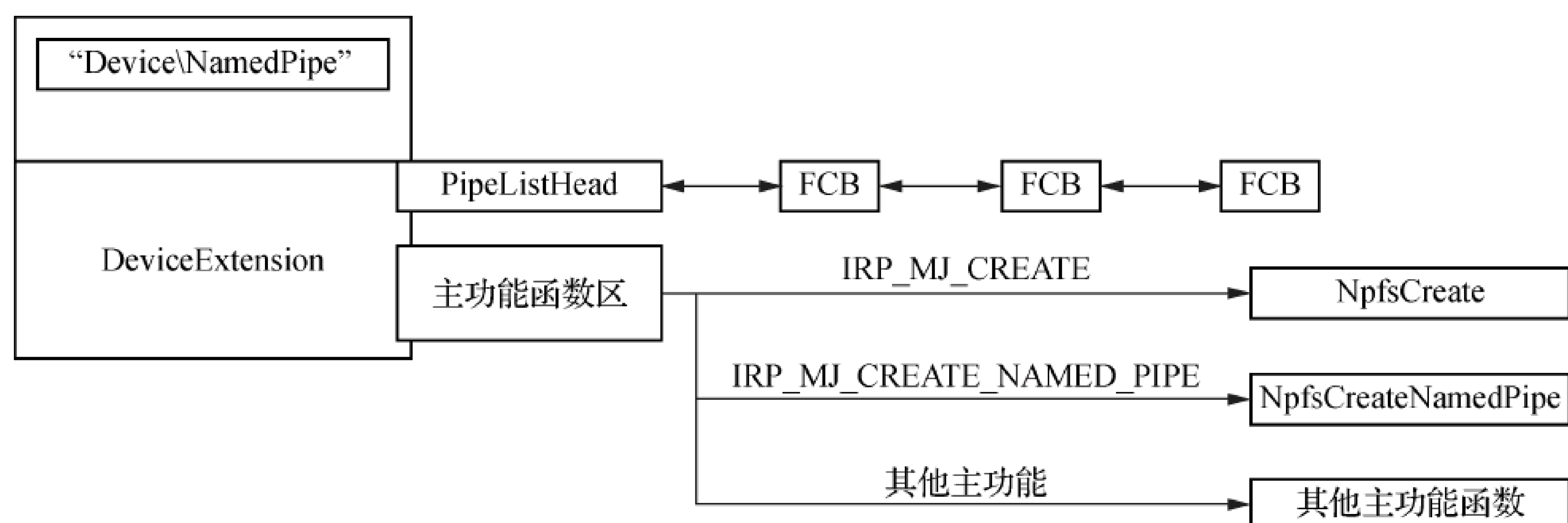


图 11－7　入口函数创建的命名管道设备对象

设备对象与设备实例的关系就好比面向对象语言中的类和对象的关系。在操作系统框架下，作为“类”的设备对象是由驱动函数的入口函数创建的；而作为“对象”的设备实例则是由 NtCreateNamedPipeFile 这样的系统调用创建的。

入口函数 DriverEntry 创建命名管道设备对象，对象的符号链接为“\Device\NamedPipe”。在设备对象的扩展结构中还有几个重要的域，包括 PipeListHead 队列和设备主功能函数。

每个创建的命名管道设备实例在设备对象中都由一个 NPFS_FCB 结构来代表一个实例，其中“NPFS”表示命名管道文件系统（Named Pipe File System），“FCB”表示文件控制块（File Control Block），因此 NPFS_FCB 就是命名管道文件控制块，这个控制块是要挂到命名管道设备对象队列中的，即全局的 PipeListHead 队列，表示在命名管道设备对象上存在了若干个实例（一个类可以有多个实例）。

主功能函数是每个驱动都具备的，这是驱动程序框架规定的。主功能函数定义了诸如设备创建、删除、读写等操作的响应函数。这是因为对于每种具体的设备，其创建和操作方式必然是不同的，内核无法具备能满足所有设备操作的功能函数，因此功能函数只能由对应设备的驱动程序自己实现。功能函数指针存放在设备对象的 MajorFunction 数组中，数组的下标称为主功能码，表示 I/O 管理器对这种设备进行的操作，函数体本身在驱动模块中。命名管道设备对象比较重要的主功能码有 IRP_MJ_CREATE_NAMED_PIPE 和 IRP_MJ_CREATE。主功能码和功能函数在后面的设备驱动相关章节中会有详细描述。

在执行命名管道设备对象的解析函数（IoParseDevice）时，会出现以下两种情况：

➢ 当遇到主功能码为 IRP_MJ_CREATE_NAMED_PIPE 的请求时，意味着设备类型是 CreateFileTypeNamedPipe，这表示创建命名管道设备对象，此时调用主功能函数 NpfsCreateNamedPipe。

➢ 当遇到主功能码为 IRP_MJ_CREATE 的请求时，意味着设备类型是 CreateFileTypeNone，这表示打开命名管道，此时调用主功能函数 NpfsCreate。

I/O 管理器每次调用 NpfsCreateNamedPipe 或 NpfsCreate 时都会以一个 FILE_OBJECT 来表示本次访问的上下文，因此也只有创建或打开这样的操作才会产生这个数据结构。一个设备实例可能会有多个 FILE_OBJECT 上下文，代表了多次的访问，例如一个管道可能打开多条连接，每一条连接就代表了一次访问。

命名管道设备实例中以上下文控制块（Context Control Block，CCB）来代表一个服务端口。在命名管道中同样有客户端和服务端之分，但无论是哪一端，要通信则必须有服务端口（CCB），有几个连接就要有几个服务端口（CCB），当然一个命名管道设备实例的 CCB 也是有上限的。可以对比 TCP 方式的连接机制，它也有端口，服务端也有端口数量上限。CCB 包含了一个读写内存缓冲区和这个缓冲区的读指针与写指针，可以看出这是个循环缓冲队列。同时 CCB 中还有个 PipeState 字段来表示状态，这些状态包括：

➢ **FILE_PIPE_DISCONNECTED_STATE**：尚未连接状态；
➢ **FILE_PIPE_LISTENING_STATE**：正在监听状态；
➢ **FILE_PIPE_CONNECTED_STATE**：连接成功状态；
➢ **FILE_PIPE_CLOSING_STATE**：连接关闭状态。

除了上述几个重要的域，还包括：

➢ **ConnectEvent**：事件对象用于阻塞正在寻找监听 CCB 的客户端线程，或者被监听方线程（服务端线程）激活以唤醒正在等待的客户端线程；
➢ **ReadEvent**：事件对象用于在读出方线程发现无数据可读时阻塞自己，或者被写入方线程激活以唤醒读出方线程继续读数据；
➢ **WriteEvent**：事件对象用于在写入方发现数据缓冲区已满时阻塞自己，或者被读出方线程激活以唤醒写入方线程继续写数据。

除此之外 CCB 还包括一个用于指明对端 CCB 的 OtherSide 域以及一些其他属性。

一个 NPFS_FCB 结构中包含一个 ServerFCBListHead 链表和一个 ClientFCBListHead 链表，分别挂载服务端 CCB 和客户端 CCB，若服务端和客户端的 CCB 彼此之间有联系，则说明两者之间已经建立了连接，如图 11－8 所示。

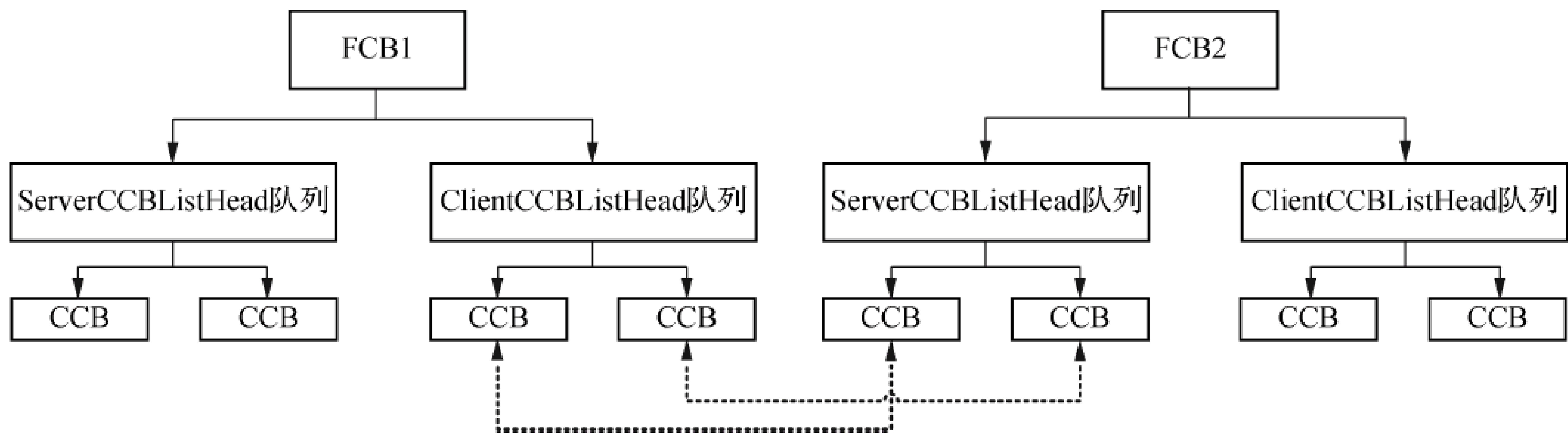


图 11－8　FCB 与 CCB 的关系

主功能函数 NpfsCreateNamedPipe 负责创建命名管道设备实例,步骤如下:

(1) 创建 NPFS_FCB 以代表一个命名管道设备实例,并将该实例挂入设备对象的 PipeListHead 队列中。

(2) 创建一个 CCB 代表服务端端口,并将该 CCB 挂入服务端的 ServerCCBListHead 队列中,同时将 CCB 设置为监听状态。

服务端监听的时候,服务端和客户端做如下交互:

➢ 在服务端 FCB 的 ClientCCBListHead 队列中扫描 CCB,若有正在等待连接的 CCB 则激活 CCB 的 ConnectEvent 事件以唤醒客户端线程。正是因为 FCB 的 ServerCCBListHead 队列中没有足够的 CCB 供客户端 CCB 来连接,才会导致客户端 CCB 未被激活,从而使客户端线程阻塞等待。

➢ 同时,将服务端 CCB 挂入 FCB 的 WaiterListHead 队列,表示正在等待被客户端 CCB 的连接。此时若有客户端 CCB 正在等待连接,则会因为线程唤醒而继续处理连接事务。

主功能函数 NpfsCreate 负责与服务端建立连接,其作用可以类比 TCP 客户端的连接建立,包括如下步骤:

(1) 寻找目标管道的 FCB,找不到则返回失败。

(2) 创建并初始化客户端 CCB,例如分配缓冲区等,此时 CCB 的状态是 FILE_PIPE_DISCONNECTED_STATE。

(3) 在 FCB 的 WaiterListHead 队列中搜索处于监听状态的 CCB(NpfsCreateNamedPipe 执行的第二步已将监听状态的 CCB 挂入 WaiterListHead 队列了)。如果搜索不到再到 ServerCCBListHead 队列中去找,找不到则出错返回。

(4) 建立连接,即将客户端 CCB 挂入 FCB 的 ClientCCBListHead 队列中,同时使两个 CCB(在服务端找到的 CCB 和客户端 CCB)的状态变成 FILE_PIPE_CONNECTED_STATE 并互相指向对方,使双方连接起来。

命名管道有两种通信模式:流模式和报文模式,前者没有边界,后者有边界。这点可以类比 TCP 和 UDP 通信协议,TCP 相当于没有边界的流模式,要通过上层协议来区分协议边界;而采用 UDP 时发送和收到的是 UDP 包,既然是包就有边界。因此在采用流模式通信时,双方的线程利用 CCB 的读、写指针在缓冲区内读写数据,只有当读、写两个指针碰在一起时才认为缓冲区为空,此时可以阻塞读线程,但是只要不同时阻塞读、写两个线程,它们就可以异步工作,彼此不依赖读的进度或写的进度,因此两个线程的通信可以看作是异步的。在采用报文模式时,双方的线程每次只交互一个报文,以保证读写的时候没有"粘包",因此它们的通信可以看作是同步的。

如图 11-9 所示是命名管道数据交互时的一般流程。该流程比较简单,就不详细描述具体细节和步骤了,大家只要明白读、写两个线程的数据交互是一个走走停停的过程就行了,写入方线程能控制读出方线程的执行,而读出方线程亦能左右写入方线程的进度。

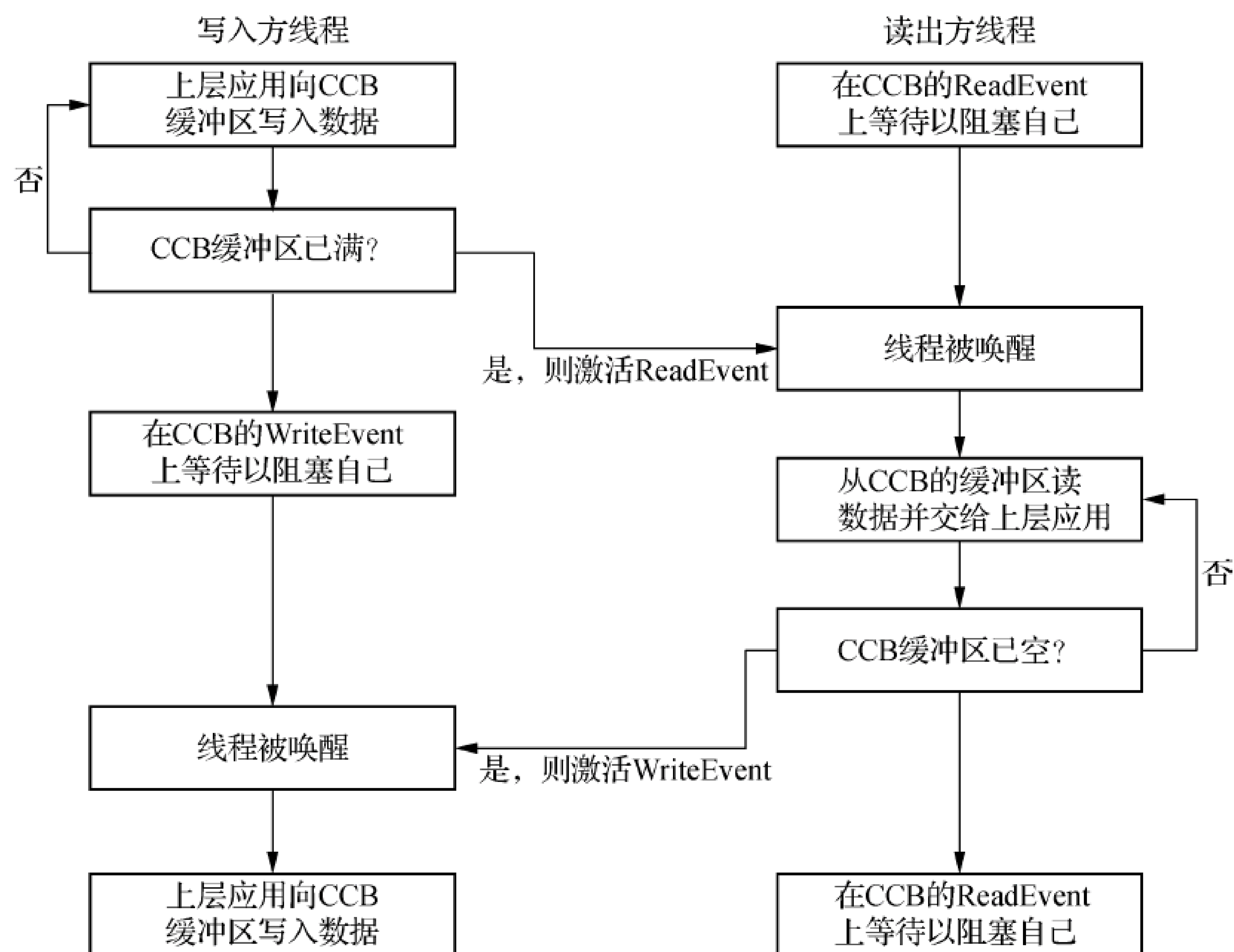


图 11－9　命名管道数据交互的一般流程

11.3　WMI

WMI 的全称是 Windows Management Instrumentation，即 Windows 管理规范，是 Windows 的基础模块。应用进程可以通过 WMI 脚本（VB 脚本）管理本地和远程计算机上的资源，例如 CPU 序列号等。WMI 屏蔽了 Windows API 不支持远程调用或脚本调用的问题，使脚本语言可以通过统一的 WMI 访问只有 API 才能访问到的资源。不过对于脚本语言来说这有个前提：**必须支持 ActiveX 技术**。

因此，WMI 更像 RPC 手段的一种，并且是专门用于获取操作系统资源的。基于此，也可以把 WMI 看作进程间通信的手段，虽然 WMI 通信既不能双向，又对交换的数据作了限定。

WMI 是对 WBEM（Web-Based Enterprise Management，基于 Web 的企业管理）模型的一种实现。WBEM 模型规范了工业界企业网络中资源的描述和使用，由 DMTF（Distributed Management Task Force，分布式管理任务组）在包括 Compaq、Sun、Microsoft 在内的许多厂商的帮助下创立，其中 CIM（Common Information Model，公共信息模型）和 MOF（Managed Object Format，托管对象格式）组件为 WMI 提供了主要支持。

CIM 用来命名计算机的物理和逻辑单元，例如磁盘分区、应用进程实例等。它是面向对象的模型，具有类和对象的概念，其中：

- 对象代表着系统里被管理的一个具体的单元。
- 类是被管理单元的模板。

➢ 命名空间是类的集合，一般每个命名空间都是面向特定的管理领域的。

CIM 分为核心模型、公共模型和扩展模型三个层次。其中核心模型包含了对所有管理领域都通用的类定义；公共模型包含了对特定管理领域通用的类定义；扩展模型包含了与 Windows 具体技术有关的类定义。

WMI 的作用一是获取信息，二是提供数据，这两个功能的提供者分别是 WMI 类（WMI Classes）和 WMI 提供者（WMI Provider）。所有的 WMI 对象都使用类 SQL 查询语言——WQL 来进行信息获取。

下面我们借用 MSDN 对于 WMI 的描述（如图 11－10 所示）来讲解 WMI 架构。

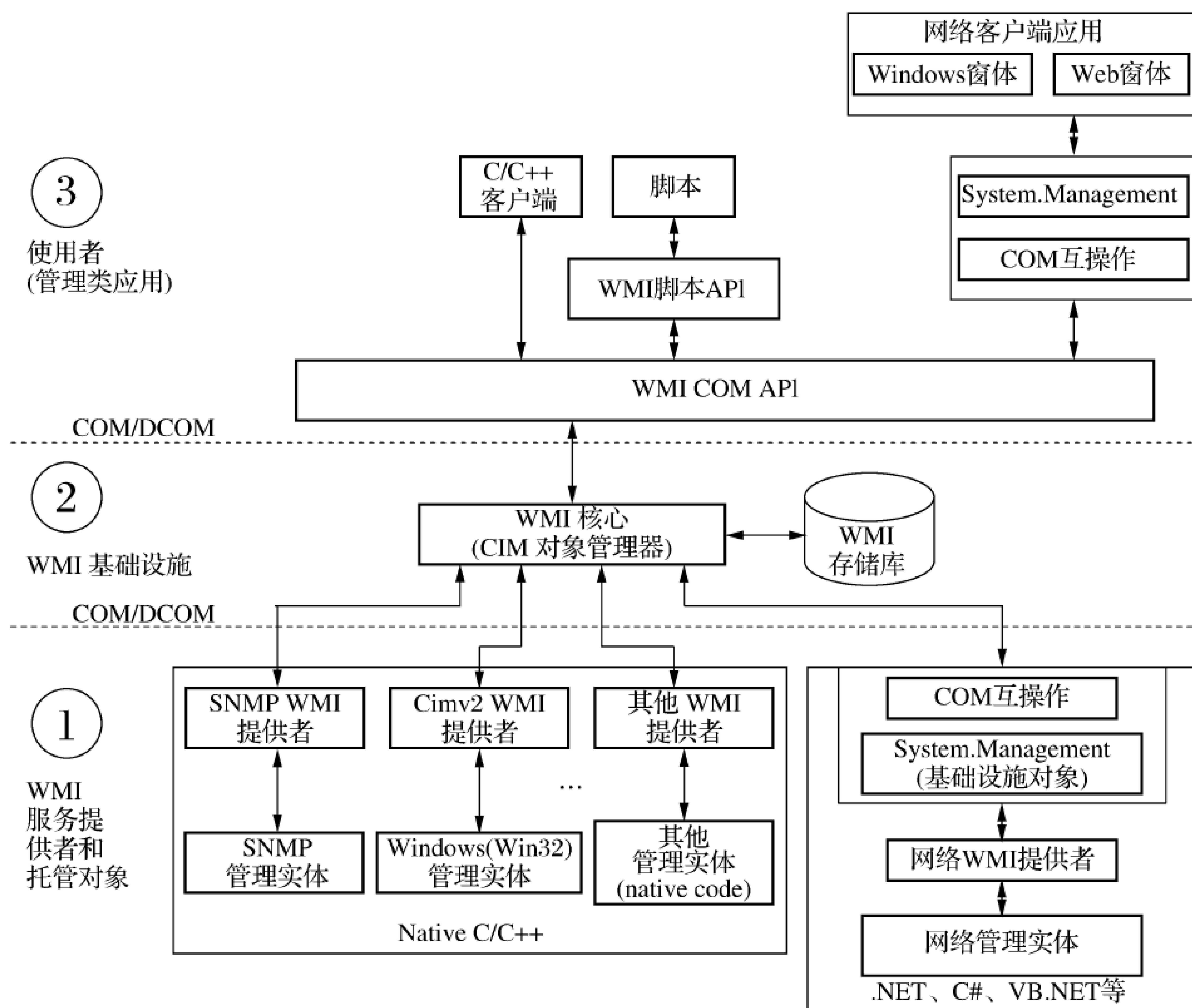


图 11－10　MSDN 对于 WMI 架构的描述

1. WMI 使用者

WMI 使用者（WMI Consumers）位于 WMI 构架的最上层，是 WMI 的载体，可以查询和枚举数据，也可以运行 WMI Provider 的方法，还可以向 WMI 订阅消息。当然这些操作都要有相应的 Provider 来提供。无论采用什么样的语言（C++、VB 脚本、. NET 等）来使用 WMI，最终都是通过 COM 接口实现了对 WMI 资源的访问。

2. WMI 基础设施

WMI 基础设施（WMI Infrastructure）是 Windows 系统的系统组件，包含两个模块：

Winmgmt(WMI 服务)和 WMI Repository(WMI 存储库)。其中 WMI 存储库是通过 WMI Namespace(WMI 命名空间)组织起来的。

系统启动时,WMI 服务会创建诸如 root\default、root\cimv2 或 root\subscription 这样的 WMI 命名空间,同时会预安装一部分 WMI 类的定义信息到这些命名空间中。其他命名空间是在操作系统或者应用进程调用相关的 WMI 提供者时才被创建出来的。此外,WMI 存储库是用于存储 WMI 静态数据的存储空间,我们获取系统资源信息都是通过 WMI 存储库来实现的。

WMI 服务扮演着 WMI 提供者、管理应用进程和 WMI 存储库之间的协调者角色。一般来说,它是通过一个共享的服务进程 svchost. exe 来实施的。当第一个管理应用向 WMI 命名空间发起连接时,WMI 服务将会启动;当管理应用不再调用 WMI 时,WMI 服务将会关闭或者进入低内存状态。

WMI 服务和上层应用进程之间是通过 COM 接口来通信的,当一个应用进程通过接口向 WMI 发起请求时,WMI 将判断该请求是请求静态数据还是动态数据:

- 如果请求的是一个静态数据,WMI 将从 WMI 存储库中查找数据并返回;
- 如果请求的是一个动态数据,比如一个托管对象的当前内存情况,WMI 服务将请求传递给已经在 WMI 服务中注册的相应的 WMI 提供者,WMI 提供者将数据返回给 WMI 服务,WMI 服务再将结果返回给请求的应用。

3. WMI 提供者

WMI 提供者(WMI Provider)是监控一个或者多个托管对象的 COM 接口。所谓托管就是任意逻辑或物理组件都可通过 WMI 进行发布和管理,包括系统、磁盘、外围设备、事件日志等方方面面的资源都可被托管。一个托管对象(Managed Object)就是一个逻辑或者物理组件,比如硬盘驱动器、网络适配器、数据库系统、操作系统、进程或者服务,在 Windows 中定义为 Win32_Process、Win32_Service、AntiVirusProduct、Win32_StartupCommand 等对象。和驱动程序相似,WMI 提供者通过托管对象向 WMI 服务提供数据,同时将 WMI 服务的请求传递给托管对象。

从文件的角度来说,WMI 提供者是由一个实现业务逻辑的 DLL 文件和承载着描述数据和操作类的托管对象格式(Managed Object Format)文件组成,两个文件都保存在% Windir% \System32\Wbem 目录下。每个 WMI 提供者都有一个 CLSID 以便在注册表中区别相关联的 COM 接口,CLSID 用于查找实现该提供者业务逻辑的实际 DLL 文件。

4. WMI 数据的远程传输

Windows 提供了两种协议进行 WMI 数据远程传输,如下所示:

- DCOM(Distributed Component Object Model,分布式组件对象模型)协议;
- WinRM(Windows Remote Management,Windows 远程管理)协议。

DCOM 以 TCP 的 135 端口作为监听端口,WinRM 采用 TCP 的 5985(HTTP)或 5986(HTTPS)作为监听端口。WinRM 是一种基于 SOAP 格式的设备管理协议,目前已取代了

DCOM 成为 Windows 主推的远程管理协议。

本章小结

本章介绍了进程间通信机制的相关概念。包括本地过程调用(LPC)、命名管道和 WMI 技术。

第12章　Windows PE 文件

所谓 Windows PE 文件，就是 Windows 可移植执行(Portable Executable)文件，前文多有提及。在 Windows 体系下，我们熟悉的 DLL、EXE、SYS、COM、OCX 等格式的文件都是 PE 文件，就像 Linux 的 SO(Shared Object，共享对象)等 ELF(Executable and Linkable Format，可执行与可链接格式)文件一样。PE 文件的历史可以追溯到 1993 年，由 Windows NT 系统引入，但是这么多年来其格式和结构也没有发生过大的变化。

PE 文件格式与 Linux 的 ELF 文件格式同宗同源，ELF 文件包括 ELF 头、程序头、区段和区段表头 4 部分，而 PE 文件则包括 DOS 头、PE 头、区段和区段表头等，两者的构成和作用都差不多，且两者都是基于 COFF 文件格式发展起来的。所谓 COFF，就是通用目标文件格式(Common Object File Format)。我们使用编译器时生成的 OBJ 文件就是 COFF 格式的。本章将按照图 12-1 所示的提纲并以作者所用计算机中的 32 位动态库为例详细考查 PE 文件的构成。在介绍 PE 文件之前不妨先来考察一下 COFF 文件。

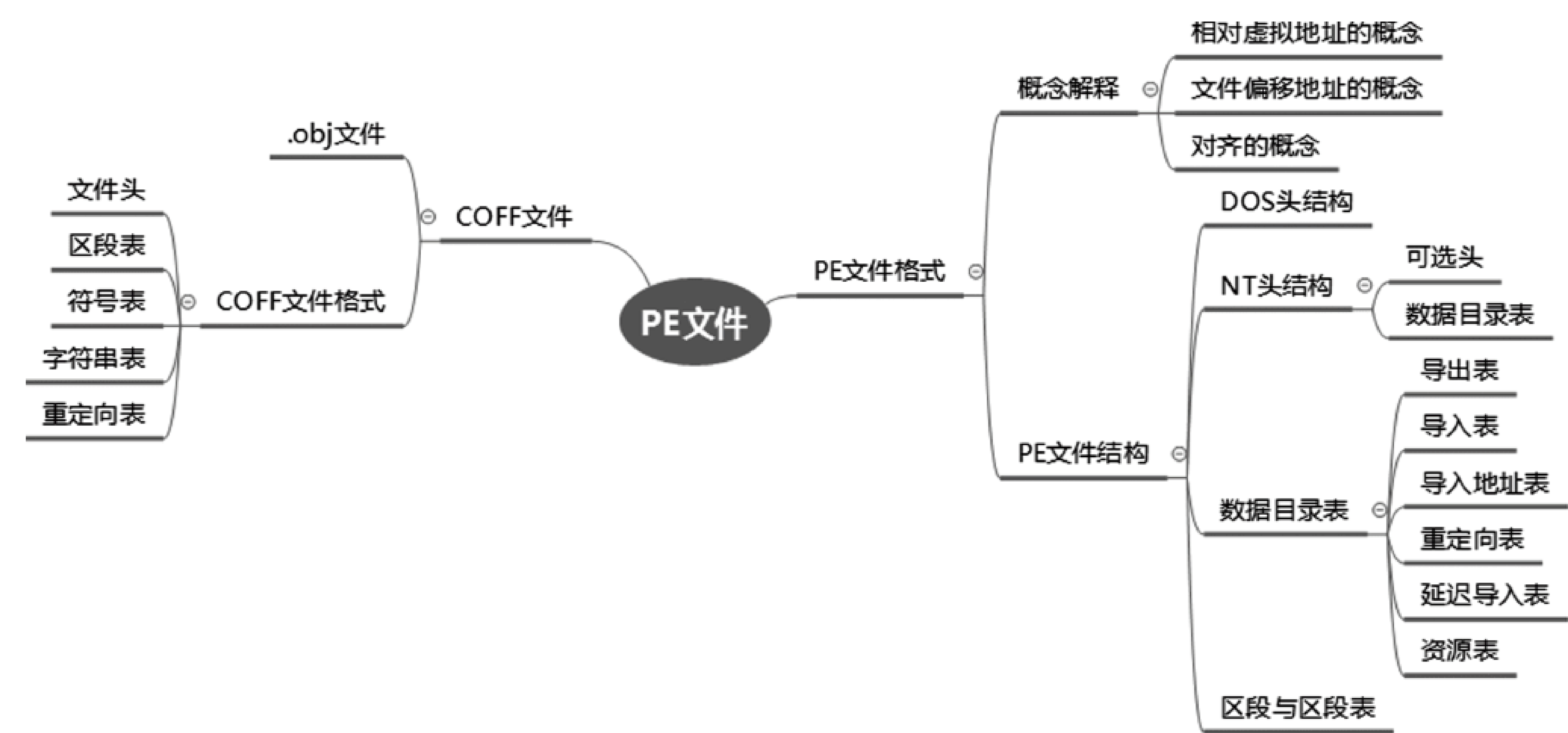


图 12-1　本章提纲

12.1　COFF 文件格式

COFF 这种二进制文件格式是用来存储符号对应的数据并组织符号之间的引用关系的。所谓符号，就是一部分二进制数据的别名，例如变量名、函数名、字符串常量等。而所谓引用关系，比如函数 A 调用了函数 B，本质上就是函数 A 的二进制代码引用了函数 B 的二进制数据。在 32 位 Windows 系统下 COFF 文件格式如图 12-2 所示。

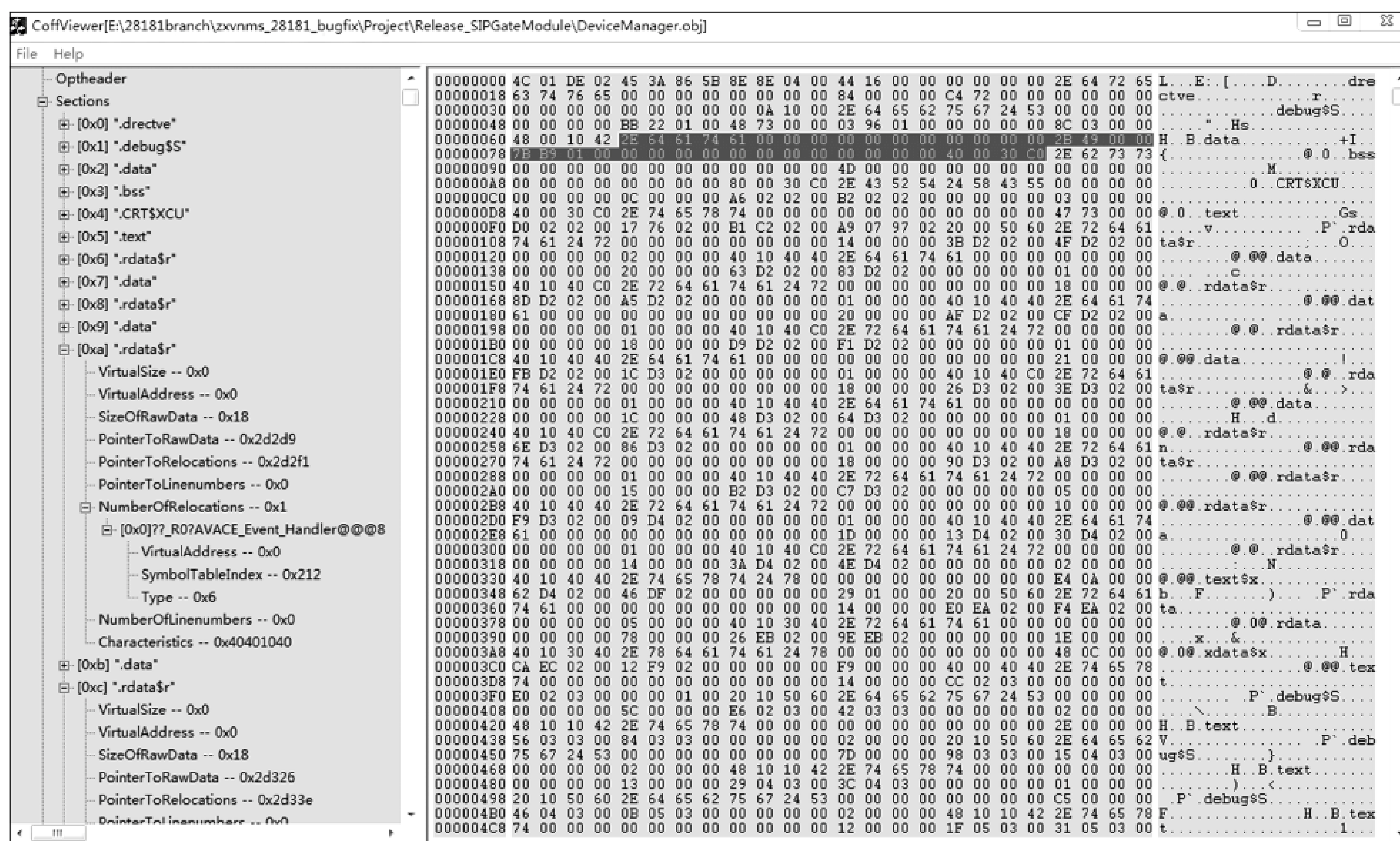


图 12－2　COFF 文件格式

在编译代码的时候，编译器将每个源文件编译成. obj 文件，一个源文件对应一个. obj 文件，而. obj 文件就是 COFF 格式的文件。但这种文件只是我们编译链接过程中的中间文件，每个文件的代码和数据并不完整，操作系统也无法解释和执行这些中间文件，因此还需要链接器将其链接为可执行文件（例如 DLL、EXE 这些格式的 PE 文件）。链接器是这样处理这些中间文件的：

① 将多个. obj 文件中相同的区段合并成一个区段，以组织成一个具有代码和数据的完整二进制文件；

② 修复重定向数据；

③ 生成针对特定处理器平台架构和操作系统的可执行文件。

COFF 文件格式包括以下结构（均以 32 位 Windows 系统为例）：

1）文件头（Header）

这是描述 COFF 文件全部信息的数据结构，因此没有文件头就无法解析 COFF 文件。文件头包含了平台名称、区段数目、符号表指针、可选头大小、符号总数量、文件标记等重要数据，以下是 Windows 对文件头的定义：

```
typedef struct _FILEHEADER
{
    unsigned short machine;                //平台名称
    unsigned short numberOfSections;       //区段数，记录该 COFF 文件一共有多少区段
    unsigned long  timeDateStamp;          //COFF 文件的创建时间
    unsigned long  pointerToSymbolTable;   //符号表在文件中的偏移
    unsigned long  numberOfSymbols;        //COFF 文件中的符号总数量
    unsigned short sizeOfOptionalHeader;   //可选头长度，但 COFF 文件没有可选头，因此为 0
    unsigned short characteristics;        //文件标记，记录着文件的属性
} FILEHEADER, * PFILEHEADER;
```

其中,平台名称的取值可以包括但不限于表 12－1 中的值。

表 12－1　COFF 文件头中的平台名称定义

平台名称定义	定义值	平台名称描述
IMAGE_FILE_MACHINE_UNKNOWN	0x0	可适用于任何平台
IMAGE_FILE_MACHINE_ALPHA	0x184	在 Alpha_AXP 处理器平台下运行
IMAGE_FILE_MACHINE_ARM	0x1C0	在 ARM 处理器平台下运行
IMAGE_FILE_MACHINE_ALPHA64	0x284	在 Alpha 64 位处理器平台下运行
IMAGE_FILE_MACHINE_I386	0x14C	在 X86 处理器平台下运行
IMAGE_FILE_MACHINE_IA64	0x200	在 X64 处理器平台下运行
IMAGE_FILE_MACHINE_MIPS16	0x268	在 MIPS 16 位处理器平台下运行
IMAGE_FILE_MACHINE_MIPSFPU	0x266	在 MIPS 带浮点处理器平台下运行
IMAGE_FILE_MACHINE_MIPSFPU16	0x366	在 MIPS 带浮点处理器平台下运行
IMAGE_FILE_MACHINE_POWERPC	0x466	在 PowerPC 处理器平台下运行
IMAGE_FILE_MACHINE_R3000	0x1F0	MIPS 平台小端序
IMAGE_FILE_MACHINE_R4000	0x162	MIPS 平台小端序
IMAGE_FILE_MACHINE_R10000	0x168	MIPS 平台小端序
IMAGE_FILE_MACHINE_SH3	0x1A2	SH3 平台小端序
IMAGE_FILE_MACHINE_SH4	0x1A6	SH4 平台小端序
IMAGE_FILE_MACHINE_THUMB	0x1C2	在 ARM_Thumb 处理器平台下运行

从上表可以看出,针对各种架构的 CPU 平台,甚至包括精简指令集的 MIPS 架构,COFF 文件的平台名称都具有相应的值表示,而 PE 文件来自于 COFF 文件,PE 文件的 NT 头中平台名称也是相同的取值,由此可见 PE 文件的“可移植性”还真不是浪得虚名。

2）区段表(Sections)

区段分为代码段(.text)、数据段(.data)、只读数据段(.rdata)、未初始化数据段(.bss)、资源段(.rsrc)、调试段(.debug)等。区段头描述区段名称、区段数据在文件中的偏移、区段符号数量、重定向数据的偏移以及行号表数量等信息,多个区段头组成了区段表。以下是 Windows 对区段头的定义:

```
typedef struct_SECTIONHEADER
{
    char            name[8];                      //区段名称,例如.text
    unsigned long   virtualSize;                  //虚拟大小,在 COFF 文件中为 0
    unsigned long   virtualAddress;               //虚拟地址,在 COFF 文件中为 0
    unsigned long   sizeOfRawData;                //区段原始数据的字节数
    unsigned long   pointerToRawData;             //区段原始数据在文件中的偏移
    unsigned long   pointerToRelocations;         //区段重定向表在文件中的偏移
    unsigned long   pointerToLinenumbers;         //行号表在文件中的偏移
    unsigned short  numberOfRelocations;          //重定向表个数
    unsigned short  numberOfLinenumbers;          //行号表个数
```

```
    unsigned long  characteristics;                    //区段标识,记录该区段的属性
} SECTIONHEADER, * SECTIONHEADER;
```

表 12 – 2 中是区段标识的取值。

表 12 – 2　COFF 区段头中的区段标识定义

区段标识定义	定义值	区段标识描述
IMAGE_SCN_CNT_CODE	0x00000020	区段包含可执行代码
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	区段包含初始化数据
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	区段包含未初始化数据
IMAGE_SCN_LNK_INFO	0x00000200	区段包含注释或其他信息，比如. drectve 区段
IMAGE_SCN_LNK_REMOVE	0x00000800	区段不会成为可执行文件的一部分
IMAGE_SCN_ALIGN_1BYTES	0x00100000	区段对齐粒度为 1 字节
IMAGE_SCN_ALIGN_4BYTES	0x00200000	区段对齐粒度为 2 字节
IMAGE_SCN_ALIGN_XBYTES	0x00400000	区段对齐粒度为 X 字节
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	区段含有外部重定向表
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	当需要时，区段可能会被丢弃
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	区段不能被加入缓存
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	区段不会被分页
IMAGE_SCN_MEM_SHARED	0x10000000	区段在内存中会被共享
IMAGE_SCN_MEM_READ	0x40000000	区段可以被读取
IMAGE_SCN_MEM_WRITE	0x80000000	区段可以被写入

3）符号表(Symbols)

符号信息用于描述符号名称、符号类型等，多个符号信息构成了符号表。符号表在 COFF 文件中相当重要，没有符号表，链接器在链接多个 COFF 文件时无法识别函数代码和全局变量保存在文件中的什么位置。Windows 定义的符号表如下所示：

```
typedef struct_SYMBOL
{
    union {
      char name[8];                   //符号名称,最大长度为 8 字节
      struct {
        unsigned long zero;           //字符串表标识,符号名超出 8 字节时为 0
        unsigned long offset;         //字符串偏移,zero 字段为 0 时保存的是字符串表的索引值 value
      } e;
    } e;
    unsigned long  value;              //符号值
    short          section;            //符号所在区段号码
    unsigned short type;               //符号类型,用于区分符号是函数还是非函数
    unsigned char  Class;              //符号存储类型,例如符号是 extern 还是 static 类型等
    unsigned char  numberOfAuxSymbols;      //符号附加记录数
} SYMBOL, * PSYMBOL;
```

4）字符串表(Strings)

字符串表用于保存字符个数超出符号表和区段头名称数组最大个数的字符串。Windows 定义的字符串表如下所示：

```
typedef struct_STRIGTABLE
{
    unsigned int Size;        //记录字符串表的所有字符的总字节数,包括 Size 本身
    char        Data[1];      //字符串表所保存的字符串
} STRIGTABLE, * PSTRIGTABLE
```

5）重定向表

重定向表的作用是指出哪些符号引用了另外的符号以及是用何种方式引用这些符号的，但并非所有的被引用符号都有重定向信息，一般情况下只有那些含有可执行属性的区段才会有重定向表，或者说只有当引用符号的位置和被引用符号的数据的位置不在同一个区段时才会产生重定向信息。Windows 定义的重定向表如下所示：

```
typedef struct_RELOCATION
{
    unsigned long virtualAddress;      //重定向数据产生的位置,也就是符号被引用的位置,是基于本区
                                       //段的原始数据的开始位置的偏移
    unsigned long symbolTableIndex;    //被引用的符号是哪一个符号,这是个从 0 开始的符号表索引
    unsigned short type;               //重定向类型,根据不同的平台有不同的类型
} RELOCATION, * PRELOCATION;
```

12.2 PE 文件格式

自被 Windows NT 引入以来，PE 文件的结构和版本基本没有变化过。对于 64 位的 Windows，PE 文件修改的地方很少，基本上是把一些 32 位的域扩展为 64 位，只对极少的域做了修改和增删，但域的语义没怎么改变。这一方面得益于 PE 文件设计之初的高瞻远瞩，另一方面也看得出 Windows 内核中关于可执行文件的架构、机制与定义的一贯性和连续性。为了区别于 32 位的 PE 文件格式，我们将 64 位的 PE 文件格式定义为 PE32 + 。

在详细介绍 PE 文件前先看看相关概念。

12.2.1 相关概念

1. 相对虚拟地址

相对虚拟地址(Relative Virual Address,RVA)表示此地址在内存中相对于基地址的偏移。每个 PE 文件都会有一个基地址 ImageBase，相对虚拟地址 = 虚拟地址 − 基址。图 12 −3 中最后一行的 ImageBase 即该 DLL 文件的基址。当然，编译出的 DLL 文件不经修改地默认都是这个基址(0x10000000)，而 EXE 文件一般是 0x00400000，但不可能每个 DLL 文件都会占据这个基址，也只有第一个加载进来且基址为 0x10000000 的模块才有幸占据这个位置，后面的模块只能由系统来分配基址。但无论分到什么地址，相对虚拟地址都是一个差值，这个差值不会随着基地址的变化而变化。

SIPGateModule.dll

Member	Offset	Size	Value	Meaning
Magic	00000118	Word	010B	PE32
MajorLinkerVersion	0000011A	Byte	06	
MinorLinkerVersion	0000011B	Byte	00	
SizeOfCode	0000011C	Dword	002DF000	
SizeOfInitializedData	00000120	Dword	001D4000	
SizeOfUninitializedData	00000124	Dword	00000000	
AddressOfEntryPoint	00000128	Dword	00294AF1	.text
BaseOfCode	0000012C	Dword	00001000	
BaseOfData	00000130	Dword	002E0000	
ImageBase	00000134	Dword	10000000	

图 12－3　DLL 文件可选头的部分域值

注意，相对虚拟地址中无论是基地址还是偏移量都是指 PE 文件在内存中而不是在磁盘文件中的地址，如图 12－4 所示。PE 文件各区段表和各种文件头在内存中和在磁盘文件中的布局是不一样的，这是因为在内存和磁盘中各区段表对齐的粒度有时是不一样的，后文会详细描述。

SIPGateModule.dll

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	002DEEF2	00001000	002DF000	00001000	00000000	00000000	0000	0000	60000020
.rdata	0005B1D8	002E0000	0005C000	002E0000	00000000	00000000	0000	0000	40000040
.data	0012EC5C	0033C000	0012C000	0033C000	00000000	00000000	0000	0000	C0000040
.rsrc	00000390	0046B000	00001000	00468000	00000000	00000000	0000	0000	40000040
.reloc	00047286	0046C000	00048000	00469000	00000000	00000000	0000	0000	42000040

图 12－4　DLL 文件区段头中的相对虚拟地址

2. 文件偏移地址

文件偏移地址（File Offset Address，FOA）表示某个位置在磁盘文件中距离文件头基址的偏移。文件偏移地址从 PE 文件的第一个字节开始计数，起始值为 0。

3. 对齐

对齐（Alignment）分为内存对齐（Section Aligment）、文件对齐（File Aligment）和资源数据对齐三种方式。

在 32 位的 Windows 系统中，一个内存页面是 4 KB，而 PE 文件头或区段在内存中必须占据完整的内存页面。例如 PE 文件头的大小为 3.5 KB，那么它在内存中会占用 4 KB，即 1 个页面，剩余的 0.5 KB 会置零；再比如代码段（.text）占用 4.1 KB，那么它在内存中将占用 8 KB，即 2 个页面，剩余 3.9 KB 会置零。这部分置零的区域叫作“空隙”，可以利用空隙来扩展区段或者增加新的区段，这也是代码打补丁时常用的做法。类似这种占据完整内存页面的方式就叫作内存对齐，内存对齐的粒度由 PE 文件头的 SectionAligment 字段定义。

在磁盘中也存在上述对齐的问题。内存中以 4 KB 为一个内存页面（32 位，其实 64 位系统中大多也是以 4 KB 作为内存页面默认大小），磁盘中则往往以 512 B（一个扇区）作为

一个磁盘页面。因此在磁盘中 PE 文件是以 512 B 为粒度对齐的。可以想象磁盘中的 PE 文件的结构要比内存中的 PE 文件紧凑不少。不过随着磁盘空间的扩展,磁盘空间不再稀缺,Windows 也可以以 4 KB 为一个磁盘页面,这样磁盘与内存的页面大小相等了,两者的空隙和布局也都会一样,许多计算自然也就方便了。磁盘页面对齐的粒度由 PE 文件头的 FileAligment 字段定义。内存页面与磁盘页面的对齐粒度如图 12-5 所示。与磁盘页面不同,资源字节码部分一般要求以双字方式对齐。

SectionAlignment	00000128	Dword	00001000
FileAlignment	0000012C	Dword	00000200

图 12-5　内存页面与磁盘页面的对齐粒度

12.2.2　PE 文件结构

PE 文件结构按照地址由低到高的顺序依次包括 DOS 头、NT 头、区段头和区段 4 部分,后面可能还混杂着重分配信息、符号表信息、行号信息以及字串表数据。PE 文件结构如图 12-6 所示,各结构之间的关系如图 12-7 所示。

File: SIPGateModule.dll
Dos 头部
Nt 头部
文件头部
可选头部
段头部 [x]
输出目录
导入目录
资源目录
重定位目录
调试目录
Address Converter
Dependency Walker
十六进制编辑
Identifier
Import Adder
Quick Disassembler
Rebuilder
Resource Editor

Member	Offset	Size	Value
e_magic	00000000	Word	5A4D
e_cblp	00000002	Word	0090
e_cp	00000004	Word	0003
e_crlc	00000006	Word	0000
e_cparhdr	00000008	Word	0004
e_minalloc	0000000A	Word	0000
e_maxalloc	0000000C	Word	FFFF
e_ss	0000000E	Word	0000
e_sp	00000010	Word	00B8
e_csum	00000012	Word	0000
e_ip	00000014	Word	0000

图 12-6　PE 文件结构视图

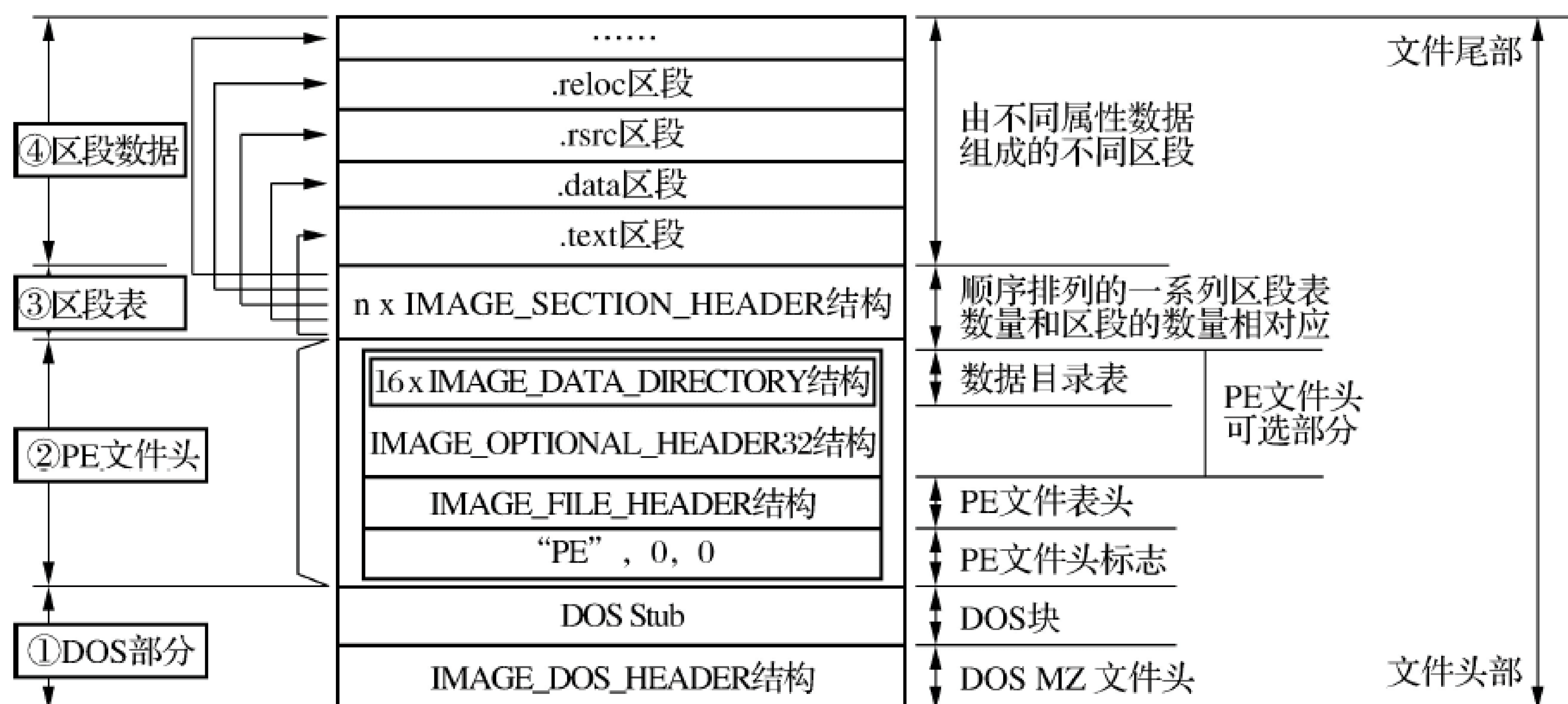


图 12-7　PE 文件各结构之间的关系

12.2.2.1　DOS 头结构

PE 文件是由一个 MS-DOS 头开始的。DOS 头是为了“兼容”16 位系统 DOS 环境而特意保留的头结构，包括了头结构本身和一段 DOS 存根程序，但这个所谓“兼容”不是说能在 DOS 下运行，而是比较友好地返回提示不能执行的消息，一般是输出“This program cannot be run in DOS mode.”这样一句提示。DOS 头结构本身的视图如图 12－8 所示。

编号	名称	偏移	大小	数据	备注
1	e_magic	00000000	2	5A4D	Magic number
2	e_cblp	00000002	2	0090	Bytes on last page of file
3	e_cp	00000004	2	0003	Pages in file
4	e_crlc	00000006	2	0000	Relocations
5	e_cparhdr	00000008	2	0004	Size of header in paragraphs
6	e_minalloc	0000000A	2	0000	Minimum extra paragraphs needed
7	e_maxalloc	0000000C	2	FFFF	Maximum extra paragraphs needed
8	e_ss	0000000E	2	0000	Initial (relative) SS value
9	e_sp	00000010	2	00B8	Initial SP value
10	e_csum	00000012	2	0000	Checksum
11	e_ip	00000014	2	0000	Initial IP value
12	e_cs	00000016	2	0000	Initial (relative) CS value
13	e_lfarlc	00000018	2	0040	File address of relocation tabl
14	e_ovno	0000001A	2	0000	Overlay number
15	e_res[0]	0000001C	2	0000	Reserved words
16	e_res[1]	0000001E	2	0000	Reserved words
17	e_res[2]	00000020	2	0000	Reserved words
18	e_res[3]	00000022	2	0000	Reserved words
19	e_oemid	00000024	2	0000	OEM identifier (for e_oeminfo)
20	e_oeminfo	00000026	2	0000	OEM information; e_oemid specifi
21	e_res2[0]	00000028	2	0000	Reserved words
22	e_res2[1]	0000002A	2	0000	Reserved words
23	e_res2[2]	0000002C	2	0000	Reserved words
24	e_res2[3]	0000002E	2	0000	Reserved words
25	e_res2[4]	00000030	2	0000	Reserved words
26	e_res2[5]	00000032	2	0000	Reserved words
27	e_res2[6]	00000034	2	0000	Reserved words
28	e_res2[7]	00000036	2	0000	Reserved words
29	e_res2[8]	00000038	2	0000	Reserved words
30	e_res2[9]	0000003A	2	0000	Reserved words
31	e_lfanew	0000003C	4	00000100	File address of new exe header

图 12－8　DOS 头结构视图

我们分别来解释其中比较重要的域。微软 SDK 或 VC 编译器目录中均含有 WinNT.h 文件，该文件中定义了一个 IMAGE_DOS_HEADER 结构体来表示 DOS 头，我们就以这个结构体的定义来列举各个域的含义。下列结构体定义中标注了部分域值的大小和偏移：

```
typedef struct_IMAGE_DOS_HEADER {
    WORD   e_magic;                  //magic number,DOS 头签名,固定是 0x5A4D
    WORD   e_cblp;
    WORD   e_cp;
    WORD   e_crlc;
    WORD   e_cparhdr;
    WORD   e_minalloc;
    WORD   e_maxalloc;
```

```
    WORD   e_ss;                       //DOS 代码的初始堆栈 SS 指针
    WORD   e_sp;                       //DOS 代码的初始堆栈 SP 指针
    WORD   e_csum;
    WORD   e_ip;                       //DOS 代码的入口指令
    WORD   e_cs;                       //DOS 代码堆栈入口
    WORD   e_lfarlc;
    WORD   e_ovno;
    WORD   e_res[4];
    WORD   e_oemid;
    WORD   e_oeminfo;
    WORD   e_res2[10];
    LONG   e_lfanew;                   //NT 头在 PE 文件中的偏移
} IMAGE_DOS_HEADER, * PIMAGE_DOS_HEADER;
```

DOS 头后面紧跟着一段 DOS 存根程序，但这段程序是由链接器定义的。当 PE 文件在 DOS 下执行时会执行这段程序，作用是在 DOS 视窗中显示“This program cannot be run in DOS mode.”这句提示，如图 12－9 所示，然后调用 int 0x21 指令（DOS 的系统中断指令）来终止程序执行。

00000040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..?.???L?Th
00000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......

图 12－9　DOS 存根程序引用的字符串

12.2.2.2　NT 头结构

PE 文件最为核心的部分就是 PE 文件头了，而核心中的核心就是 PE 文件头最后一个数据结构——可选头结构 IMAGE_OPTIONAL_HEADER32（64 位系统下为 IMAGE_OPTIONAL_HEADER64 结构），因为可选头定义了运行基址、入口地址等重要信息以及紧跟其后的目录表，没有可选头就无法找到这些重要信息，更无法找到导入表、导出表等这些运行时必需的表。

1. 可选头结构

在 WinNT.h 中定义了 IMAGE_NT_HEADER 结构来表示可选头，可选头也被称为 NT 头，32 位 Windows 定义的 NT 头结构如下所示：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, * PIMAGE_NT_HEADERS32;
```

64 位 Windows 定义的 NT 头如下所示：

```
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, * PIMAGE_NT_HEADERS64;
```

我们先来看 NT 头中前两个数据结构。Signature 即 PE 文件签名，其值固定为 0x45500000，表示“PE..”这几个 ASCII 码字母。

接下来的 IMAGE_FILE_HEADER 结构如图 12－10 所示。

编号	名称	偏移	大小	数据	备注
1	Machine	00000104	2	014C	Intel 386
2	NumberOfSections	00000106	2	0005	
3	TimeDateStamp	00000108	4	5B6C2160	2018-8-9 19:11:28
4	PointerToSymbolTable	0000010C	4	00000000	
5	NumberOfSymbols	00000110	4	00000000	
6	SizeOfOptionalHeader	00000114	2	00E0	
7	Characteristics	00000116	2	210E	dll

图 12－10　IMAGE_FILE_HEADER 结构视图

我们在 WinNT. h 中也能找到其定义：

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;                //此处与 COFF 文件的 FILEHEADER 中的 Machine 取值一致,表
            //示 X86 平台
    WORD    NumberOfSections;       //区段数量,例如.text、.data 等区段的数量
    DWORD   TimeDateStamp;          //此 PE 文件的创建时间
    DWORD   PointerToSymbolTable;   //指向 COFF 符号的指针,这是程序调试信息
    DWORD   NumberOfSymbols;        //符号数
    WORD    SizeOfOptionalHeader;   //可选头长度,即 IMAGE_OPTIONAL_HEADER 的长度
    WORD    Characteristics;        //表示文件属性,例如 IMAGE_FILE_DLL 属性
} IMAGE_FILE_HEADER, * PIMAGE_FILE_HEADER;
```

而关于 Characteristics 的取值,WinNT. h 中也有定义,如图 12－11 所示,这些值可以取并集。

```
#define IMAGE_FILE_RELOCS_STRIPPED           0x0001  // Relocation info stripped from file.
#define IMAGE_FILE_EXECUTABLE_IMAGE          0x0002  // File is executable  (i.e. no unresolved externel references).
#define IMAGE_FILE_LINE_NUMS_STRIPPED        0x0004  // Line nunbers stripped from file.
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED       0x0008  // Local symbols stripped from file.
#define IMAGE_FILE_AGGRESIVE_WS_TRIM         0x0010  // Agressively trim working set
#define IMAGE_FILE_LARGE_ADDRESS_AWARE       0x0020  // App can handle >2gb addresses
#define IMAGE_FILE_BYTES_REVERSED_LO         0x0080  // Bytes of machine word are reversed.
#define IMAGE_FILE_32BIT_MACHINE             0x0100  // 32 bit word machine.
#define IMAGE_FILE_DEBUG_STRIPPED            0x0200  // Debugging info stripped from file in .DBG file
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP   0x0400  // If Image is on removable media, copy and run from the swap file.
#define IMAGE_FILE_NET_RUN_FROM_SWAP         0x0800  // If Image is on Net, copy and run from the swap file.
#define IMAGE_FILE_SYSTEM                    0x1000  // System File.
#define IMAGE_FILE_DLL                       0x2000  // File is a DLL.
#define IMAGE_FILE_UP_SYSTEM_ONLY            0x4000  // File should only be run on a UP machine
#define IMAGE_FILE_BYTES_REVERSED_HI         0x8000  // Bytes of machine word are reversed.

#define IMAGE_FILE_MACHINE_UNKNOWN           0
#define IMAGE_FILE_MACHINE_I386              0x014c  // Intel 386.
#define IMAGE_FILE_MACHINE_R3000             0x0162  // MIPS little-endian, 0x160 big-endian
#define IMAGE_FILE_MACHINE_R4000             0x0166  // MIPS little-endian
#define IMAGE_FILE_MACHINE_R10000            0x0168  // MIPS little-endian
#define IMAGE_FILE_MACHINE_WCEMIPSV2         0x0169  // MIPS little-endian WCE v2
#define IMAGE_FILE_MACHINE_ALPHA             0x0184  // Alpha_AXP
#define IMAGE_FILE_MACHINE_SH3               0x01a2  // SH3 little-endian
#define IMAGE_FILE_MACHINE_SH3DSP            0x01a3
#define IMAGE_FILE_MACHINE_SH3E              0x01a4  // SH3E little-endian
#define IMAGE_FILE_MACHINE_SH4               0x01a6  // SH4 little-endian
#define IMAGE_FILE_MACHINE_SH5               0x01a8  // SH5
#define IMAGE_FILE_MACHINE_ARM               0x01c0  // ARM Little-Endian
#define IMAGE_FILE_MACHINE_THUMB             0x01c2
#define IMAGE_FILE_MACHINE_AM33              0x01d3
#define IMAGE_FILE_MACHINE_POWERPC           0x01F0  // IBM PowerPC Little-Endian
#define IMAGE_FILE_MACHINE_POWERPCFP         0x01f1
#define IMAGE_FILE_MACHINE_IA64              0x0200  // Intel 64
#define IMAGE_FILE_MACHINE_MIPS16            0x0266  // MIPS
#define IMAGE_FILE_MACHINE_ALPHA64           0x0284  // ALPHA64
#define IMAGE_FILE_MACHINE_MIPSFPU           0x0366  // MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU16         0x0466  // MIPS
#define IMAGE_FILE_MACHINE_AXP64             IMAGE_FILE_MACHINE_ALPHA64
#define IMAGE_FILE_MACHINE_TRICORE           0x0520  // Infineon
#define IMAGE_FILE_MACHINE_CEF               0x0CEF
#define IMAGE_FILE_MACHINE_EBC               0x0EBC  // EFI Byte Code
#define IMAGE_FILE_MACHINE_AMD64             0x8664  // AMD64 (K8)
#define IMAGE_FILE_MACHINE_M32R              0x9041  // M32R little-endian
#define IMAGE_FILE_MACHINE_CEE               0xC0EE
```

图 12－11　WinNT. h 中关于 Characteristics 取值的定义

我们重点来看 NT 可选头结构 IMAGE_OPTIONAL_HEADER32，其结构如图 12－12 所示。

编号	名称	偏移	大小	数据	备注
1	Magic	00000118	2	010B	PE32
2	MajorLinkerVersion	0000011A	1	06	
3	MinorLinkerVersion	0000011B	1	00	
4	SizeOfCode	0000011C	4	002DF000	
5	SizeOfInitializedData	00000120	4	001D4000	
6	SizeOfUninitializedDat	00000124	4	00000000	
7	AddressOfEntryPoint	00000128	4	00294AF1	File Offset:00294AF1
8	BaseOfCode	0000012C	4	00001000	
9	BaseOfData	00000130	4	002E0000	
10	ImageBase	00000134	4	10000000	
11	SectionAlignment	00000138	4	00001000	
12	FileAlignment	0000013C	4	00001000	
13	MajorOperatingSystemVe	00000140	2	0004	
14	MinorOperatingSystemVe	00000142	2	0000	
15	MajorImageVersion	00000144	2	0000	
16	MinorImageVersion	00000146	2	0000	
17	MajorSubsystemVersion	00000148	2	0004	
18	MinorSubsystemVersion	0000014A	2	0000	
19	Win32VersionValue	0000014C	4	00000000	
20	SizeOfImage	00000150	4	004B4000	
21	SizeOfHeaders	00000154	4	00001000	
22	CheckSum	00000158	4	00000000	
23	Subsystem	0000015C	2	0002	Windows GUI
24	DllCharacteristics	0000015E	2	0000	
25	SizeOfStackReserve	00000160	4	00100000	
26	SizeOfStackCommit	00000164	4	00001000	
27	SizeOfHeapReserve	00000168	4	00100000	
28	SizeOfHeapCommit	0000016C	4	00001000	
29	LoaderFlags	00000170	4	00000000	
30	NumberOfRvaAndSizes	00000174	4	00000010	

图 12－12　可选头结构视图

在 WinNT.h 中可选头的定义和解释如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    ==========Standard fields==========
    WORD     Magic;                     //PE 文件头签名,固定为 0x010B,在 64 位系统下为
                                        //0x020B
    BYTE     MajorLinkerVersion;        //链接该 PE 文件的链接器主版本号
    BYTE     MinorLinkerVersion;        //链接该 PE 文件的链接器子版本号
    DWORD    SizeOfCode;                //所有代码段(IMAGE_SCN_CNT_CODE 属性)的总大小
    DWORD    SizeOfInitializedData;     //所有已初始化数据段的总大小
    DWORD    SizeOfUninitializedData;   //所有未初始化数据段的总大小,但链接器一般将未初
                                        //始化数据段附加到常规数据段的末尾,因此该域值总
                                        //为 0
    DWORD    AddressOfEntryPoint;       //PE 文件入口相对地址,但该域一般指向运行时库代码
```

```
    DWORD   BaseOfCode;                    //代码段在内存中的相对地址
    DWORD   BaseOfData;                    //数据段在内存中的相对地址
    ==========NT additional fields==========
    DWORD   ImageBase;                     //PE 文件在内存中的首选加载地址,EXE 文件为
                                           //0x00400000,DLL 文件为 0x10000000
    DWORD   SectionAlignment;              //PE 文件加载到内存后的对齐粒度
    DWORD   FileAlignment;                 //PE 文件在磁盘中的对齐粒度
    WORD    MajorOperatingSystemVersion;   //操作系统的主版本号
    WORD    MinorOperatingSystemVersion;   //操作系统的子版本号
    WORD    MajorImageVersion;             //该 PE 文件的主版本号
    WORD    MinorImageVersion;             //该 PE 文件的子版本号
    WORD    MajorSubsystemVersion;         //已废弃
    WORD    MinorSubsystemVersion;         //已废弃
    DWORD   Win32VersionValue;             //已废弃
    DWORD   SizeOfImage;                   //PE 文件加载到内存后的映像大小,是内存对齐值的整
                                           //数倍
    DWORD   SizeOfHeaders;                 //DOS 头、NT 头和区段表的总大小,是磁盘文件对齐值
                                           //的整数倍
    DWORD   CheckSum;                      //PE 映像的校验和
    WORD    Subsystem;                     //PE 文件所属的子系统,常用的有如下几种:
                                           //IMAGE_SUBSYSTEM_NATIVE:不需要子系统
                                           //IMAGE_SUBSYSTEM_WINDOWS_GUI:使用 GUI 子系统
                                           //IMAGE_SUBSYSTEM_WINDOWS_CUI:控制台程序
    WORD    DllCharacteristics;            //DLL 特性
    DWORD   SizeOfStackReserve;            //PE 文件为线程保留的堆栈大小,默认是 1MB
    DWORD   SizeOfStackCommit;             //PE 文件为堆栈初始提交的内存大小,默认是 4KB
    DWORD   SizeOfHeapReserve;             //PE 文件为进程堆保留的内存大小,默认是 1MB
    DWORD   SizeOfHeapCommit;              //PE 文件为进程堆初始提交的内存大小,默认是 4KB
    DWORD   LoaderFlags;                   //不使用,须置为 0
    DWORD   NumberOfRvaAndSizes;           //包含了 DataDirectory 数组的元素个数
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; //数据目录表
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

其中,Subsystem 和 DllCharacteristics 的域值如图 12－13 所示。

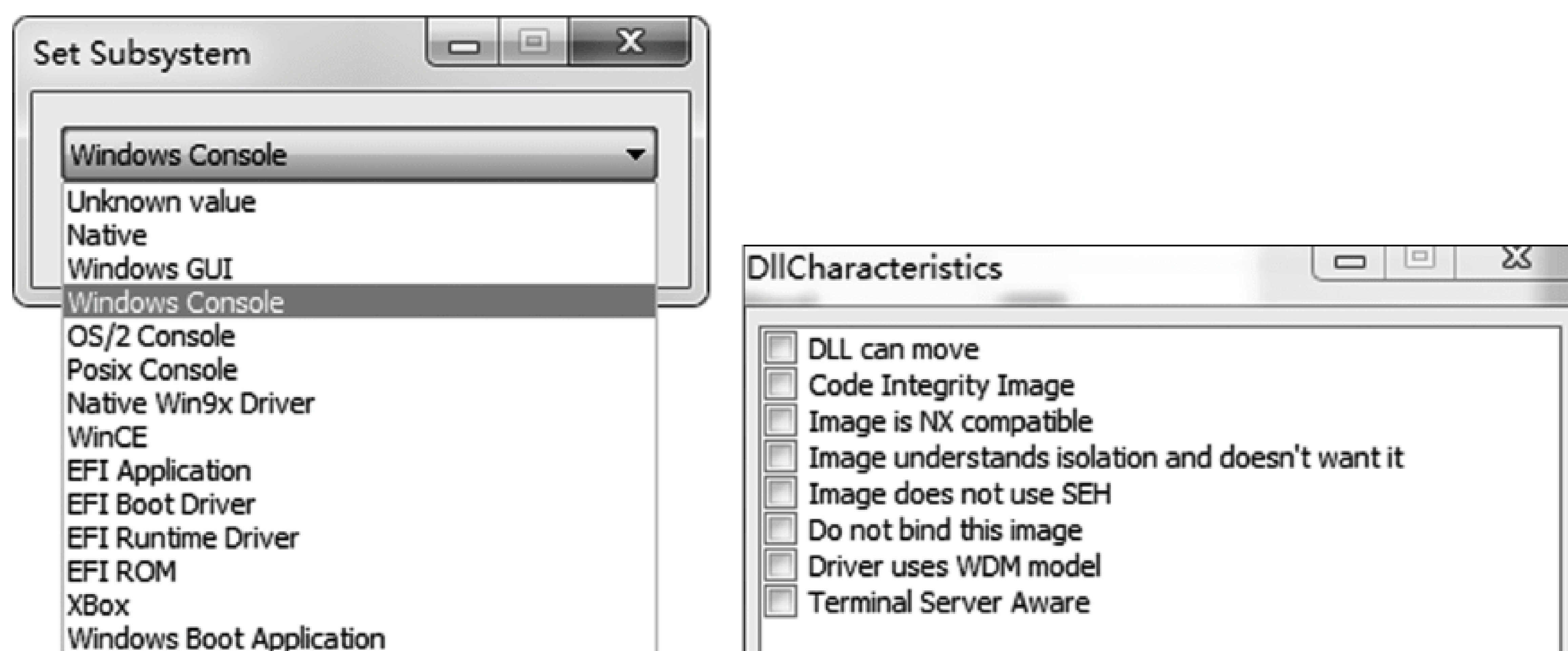


图 12－13　Subsystem 和 DllCharacteristics 的域值

2. 数据目录表

数据目录表在 NT 可选头结构中是最后一个域,它是一个 IMAGE_DATA_DIRECTORY 数据结构的数组,描述了 PE 用到的各个表在内存中的位置偏移和大小。在 32 位系统下每个 IMAGE_DATA_DIRECTORY 包含了 VirtualAddress 和 Size 这两个双字大小域值共 8 个字节,而整个数组的元素数量固定为 16,其元素包括导出表目录、导入表、导入地址表目录等,最后也包括了一个 8 字节零值的终止符。目录表结构如图 12－14 所示,它是将 IMAGE_DATA_DIRECTORY 拆分成 RVA 和 Size 两个值的结构表。

从图 12－14 可见，数据目录表中包含了以下数据表的 RVA 和大小：导出表、导入表、资源表、异常处理表、WIN_CERTIFICATE 结构列表、重定向表、调试表、IMAGE_ARCHITECTURE_HEADER 结构数组、线程本地存储表、配置表、绑定导入表、导入地址表、延迟导入表和一个. NET 信息的最高级别信息表。在本节我们重点讲述导出表、导入表、资源表、重定向表和导入地址表，这也是我们在加载和运行 PE 模块时最常用到的。

SIPGateModule.dll

Member	Offset	Size	Value	Section
Export Directory RVA	00000178	Dword	0033B160	.rdata
Export Directory Size	0000017C	Dword	00000078	
Import Directory RVA	00000180	Dword	00333BF8	.rdata
Import Directory Size	00000184	Dword	00000140	
Resource Directory RVA	00000188	Dword	0046B000	.rsrc
Resource Directory Size	0000018C	Dword	00000390	
Exception Directory RVA	00000190	Dword	00000000	
Exception Directory Size	00000194	Dword	00000000	
Security Directory RVA	00000198	Dword	00000000	
Security Directory Size	0000019C	Dword	00000000	
Relocation Directory RVA	000001A0	Dword	0046C000	.reloc
Relocation Directory Size	000001A4	Dword	0004322C	
Debug Directory RVA	000001A8	Dword	002E0B30	.rdata
Debug Directory Size	000001AC	Dword	0000001C	
Architecture Directory RVA	000001B0	Dword	00000000	
Architecture Directory Size	000001B4	Dword	00000000	
Reserved	000001B8	Dword	00000000	
Reserved	000001BC	Dword	00000000	
TLS Directory RVA	000001C0	Dword	00000000	
TLS Directory Size	000001C4	Dword	00000000	
Configuration Directory RVA	000001C8	Dword	00000000	
Configuration Directory Size	000001CC	Dword	00000000	
Bound Import Directory RVA	000001D0	Dword	00000000	
Bound Import Directory Size	000001D4	Dword	00000000	
Import Address Table Directory RVA	000001D8	Dword	002E0000	.rdata
Import Address Table Directory Size	000001DC	Dword	00000B30	
Delay Import Directory RVA	000001E0	Dword	00000000	
Delay Import Directory Size	000001E4	Dword	00000000	
.NET MetaData Directory RVA	000001E8	Dword	00000000	
.NET MetaData Directory Size	000001EC	Dword	00000000	

图 12－14　数据目录表结构视图

12.2.2.3　数据目录表

1. 导出表

导出表是数据目录表中第一个出现的表，其 RVA 是 0x0033B160，如果加上 PE 基址，则在不考虑重定向的情况下导出表位于虚拟地址为 0x1033B160 的位置，如图 12－15 所示。

编号	名称	偏移	大小	数据	备注
1	Characteristics	0033B160	4	00000000	
2	TimeDateStamp	0033B164	4	5B6C2160	2018-8-9 19:11:28
3	MajorVersion	0033B168	2	0000	
4	MinorVersion	0033B16A	2	0000	
5	Name	0033B16C	4	0033B1A6	SIPGateModule.dll
6	Base	0033B170	4	00000001	
7	NumberOfFunctions	0033B174	4	00000003	
8	NumberOfNames	0033B178	4	00000003	
9	AddressOfFunctions	0033B17C	4	0033B188	File Offset:0033B188
10	AddressOfNames	0033B180	4	0033B194	File Offset:0033B194
11	AddressOfNameOrdinals	0033B184	4	0033B1A0	File Offset:0033B1A0

图 12－15　导出表属性视图

我们使用工具查看导出表，其位于磁盘文件（磁盘文件基址为 0）上偏移也为 0x0033B160 的位置上，这是与内存位置相呼应的。当前的导出表一共导出了 3 个函数，如图 12－16 所示。

```
.reloc                  0033B130  16 01 47 65 74 46 75 6C 6C 50 61 74 68 4E 61 6D  ..GetFullPathNam
Directorys              0033B140  65 41 00 00 04 01 47 65 74 44 72 69 76 65 54 79  eA....GetDriveTy
  Export Directory      0033B150  70 65 41 00 00 00 00 00 00 00 00 00 00 00 00 00  peA.............
    1-CommandCALL       0033B160  00 00 00 00 60 21 6C 5B 00 00 00 00 A6 B1 33 00  ....`!l[....?.
    2-Init              0033B170  01 00 00 00 03 00 00 00 03 00 00 00 88 B1 33 00  ..............?.
    3-SIPGATE_Module    0033B180  94 B1 33 00 A0 B1 33 00 8C 27 21 00 68 27 21 00  ?.?.?!.h'!.
  Import Directory      0033B190  5E 27 21 00 B8 B1 33 00 C4 B1 33 00 C9 B1 33 00  ^'!.?.?.?.
    TCPModule.dll       0033B1A0  00 00 01 00 02 00 53 49 50 47 61 74 65 4D 6F 64  ......SIPGateMod
    ACE_znv.dll         0033B1B0  75 6C 65 2E 64 6C 6C 00 43 6F 6D 6D 61 6E 64 43  ule.dll.CommandC
    ZxLog.dll           0033B1C0  41 4C 4C 00 49 6E 69 74 00 53 49 50 47 41 54 45  ALL.Init.SIPGATE
```

图 12－16　导出表的函数导出视图

导出表的结构在 WinNT.h 中也有定义，即 IMAGE_EXPORT_DIRECTORY 结构体，其域值的含义如下：

- **Characteristics**：未用到，一般为 0。
- **TimeDateStamp**：链接器生成导出表的时间。
- **MajorVersion**：未用到，一般为 0。
- **MinorVersion**：未用到，一般为 0。
- **Name**：模块名称的 RVA，指向例如“SIPGateModule.dll”这样的字符串。
- **Base**：导出函数的序号以该值为基数向上递增。
- **NumberOfFunctions**：所有导出函数的数量。
- **NumberOfNames**：按函数名导出函数的数量。
- **AddressOfFunctions**：指向一个 DWORD 数组，数组中的每一项都是一个导出函数的 RVA，顺序与导出序号（AddressOfNameOrdinals 数组中的每一项）匹配。AddressOfFunctions 本身是一个 RVA。
- **AddressOfNames**：指向一个 DWORD 数组，数组中的每一项仍然是一个 RVA，指向函数名。AddressOfNames 本身是一个 RVA。
- **AddressOfNameOrdinals**：指向一个 WORD 数组，数组中的每一项与 AddressOfNames 中的每一项对应，表示该名称的函数在 AddressOfFunctions 中的序号。

AddressOfNameOrdinals 本身是一个 RVA。

这几个域值的关系如图 12－17 所示。简单来说，AddressOfFunctions 指向导出函数地址，AddressOfNameOrdinals 指向导出函数序号（以 Base 为起始值），AddressOfNames 指向导出函数名称的地址。有的函数只导出了序号和地址而没有导出名称，因此 AddressOfNames 有可能与其他两者不一一对应。

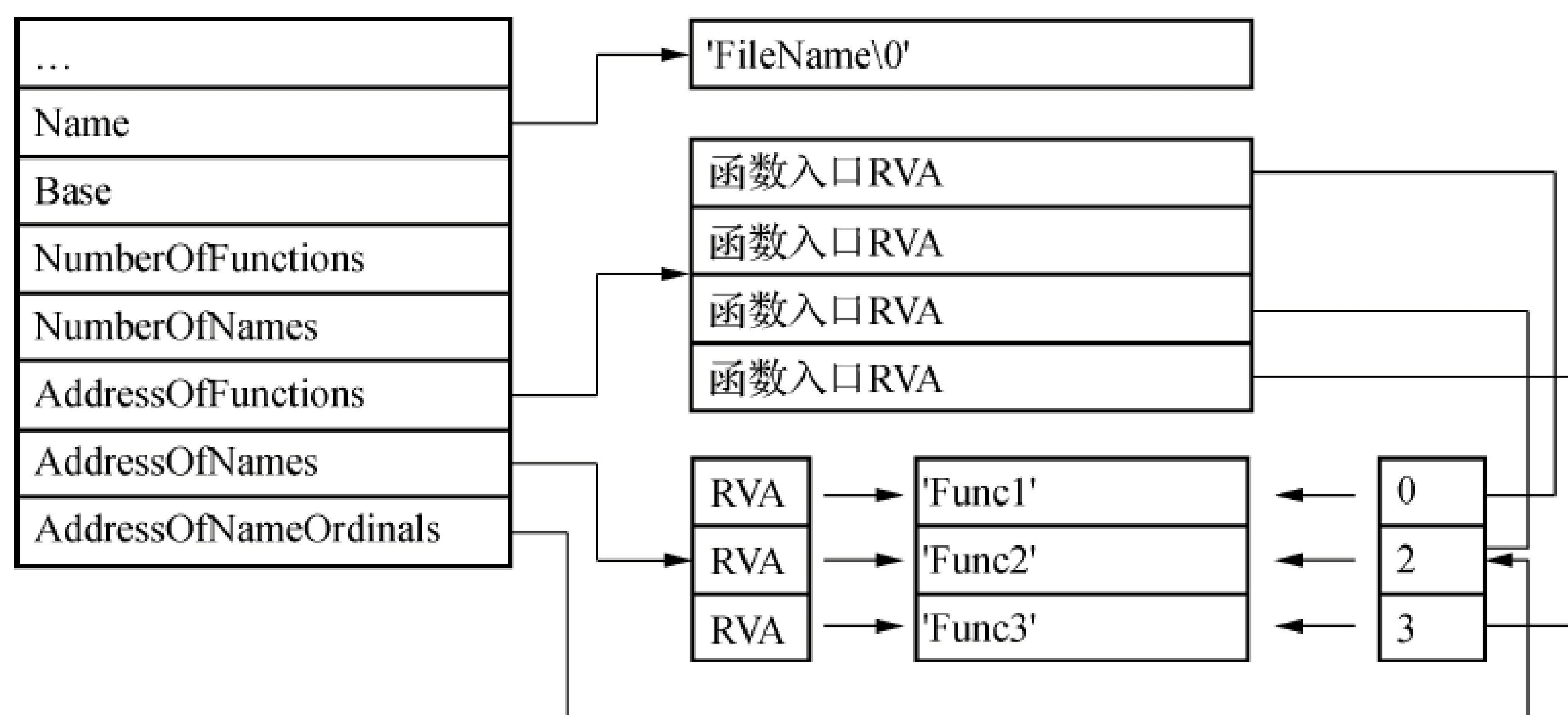


图 12－17　导出函数的地址、名称、序号三者之间的关系

编号	名称	偏移	大小	数据	备注
1	Address	0033B18C	4	00212768	File Offset:00212768
2	Ordinal	0033B1A2	2	0001	SN:2
3	Name	0033B198	4	0033B1C4	File Offset:0033B1C4 (Init)

编号	名称	偏移	大小	数据	备注
1	Address	0033B190	4	0021275E	File Offset:0021275E
2	Ordinal	0033B1A4	2	0002	SN:3
3	Name	0033B19C	4	0033B1C9	File Offset:0033B1C9 (SIPGATE_Mo(

图 12－18　导出函数地址、名称、序号三者关系示例

2. 导入表

导入表具有两层含义：该 PE 文件引用了哪些其他的 PE 文件以及该 PE 文件调用了这些文件的哪些接口。导入表结构如图 12－19 所示。

编号	名称	偏移	大小	数据	备注
1	OriginalFirstThunk	00333C20	4	003347B4	File Offset:003347B4
2	TimeDateStamp	00333C24	4	00000000	
3	ForwarderChain	00333C28	4	00000000	
4	Name	00333C2C	4	00337E8E	File Offset:00337E8E
5	FirstThunk	00333C30	4	002E0A7C	File Offset:002E0A7C

图 12－19　导入表结构视图

导入数据保存在导入表中,针对每个被导入的 PE 文件都有一张导入表,这些导入表按序排列在一起。以图 12－19 为例,我们要导入 TCPModule. dll、ZxLog. dll 等动态链接库,每个库对应一张导入表(图 12－19 中左下方)。导入表在 WinNT. h 中也有定义,即 IMAGE_IMPORT_DESCRIPTOR 结构(如图 12－20 所示),其具体域值的含义如下:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\time stamp
                                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

图 12－20 IMAGE_IMPORT_DESCRIPTOR 在 WinNT. h 中的定义

➢ **Characteristics 和 OriginalFirstThunk**:一个联合体,不为 0 时 OriginalFirstThunk 保存一个 RVA,指向 IMAGE_THUNK_DATA 数组,这个数组中的每一项表示一个导入函数。

➢ **TimeDateStamp**:映像绑定前该值为 0,绑定后为导入模块的时间戳。

➢ **ForwarderChain**:转发链,若没有转发器,这个值是－1。

➢ **Name**:被导入模块名称的 RVA,指向例如“TCPModule. dll”这样的字符串,所以一个 IMAGE_IMPORT_DESCRIPTOR 只能描述一个导入的 DLL。

➢ **FirstThunk**:指向一个 IMAGE_THUNK_DATA 数组,后文将详细说明,FirstThunk 是一个 RVA。

OriginalFirstThunk 与 FirstThunk 都指向 IMAGE_THUNK_DATA 数组,那么两者有什么区别呢? IMAGE_THUNK_DATA 结构如下所示:

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;      //PBYTE
        DWORD Function;             //PDWORD,函数地址
        DWORD Ordinal;              //按序号导入时使用
        DWORD AddressOfData;        //PIMAGE_IMPORT_BY_NAME,若按函数名导入则指向函数名
    } u1;
} IMAGE_THUNK_DATA32;
typedef IMAGE_THUNK_DATA32 * PIMAGE_THUNK_DATA32;
```

其中,Ordinal 与 AddressOfData 指向同一内存空间,这个空间大小都是 4 字节 32 位的,因此以最高位是否为 0 来区别到底是使用 Ordinal 还是 AddressOfData,为 1 就表示按序号导入,使用 Ordinal;为 0 就表示按函数名导入,使用 AddressOfData。虽然 AddressOfData 和 Ordinal 可用的位数只有 31 位,但是在一个 PE 文件中内存空间的搜索范围既不会超过 2GB,导出的序号也不会超过 2^{31},因此 31 位已经足够用了。

我们来看 OriginalFirstThunk 与 FirstThunk 之间的协作关系。在加载之前,两者分别指向两

个 IMAGE_THUNK_DATA 数组,但这个数组的 Ordinal 或 AddressOfData 却分别指向同一份内容,即编号和函数名信息,这些信息称为 INT(Import Name Table,导入名称表),如图 12-21 所示。

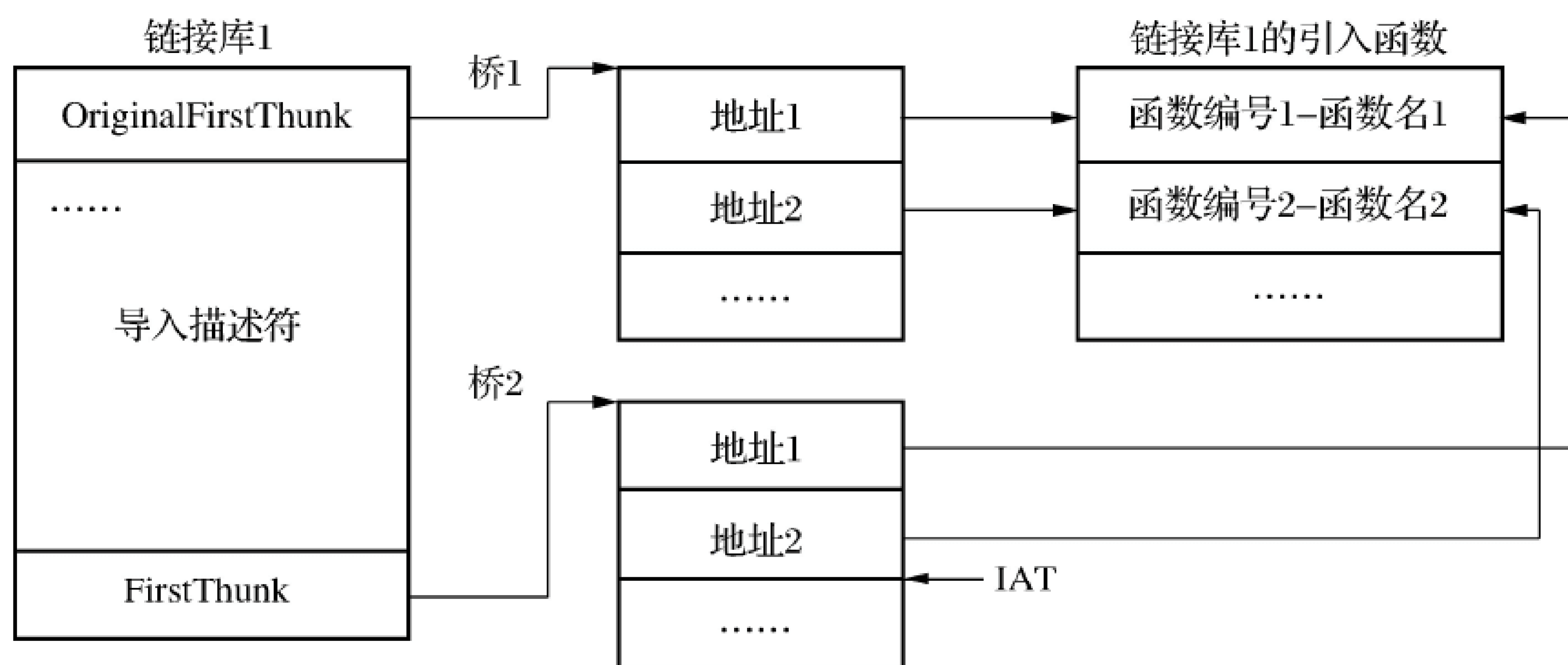


图 12-21 加载 DLL 之前 IMAGE_THUNK_DATA 数组的指向

加载之后,FirstThunk 指向的 IMAGE_THUNK_DATA 数组的 Function 域值生效,被替换成导入函数在内存中的真实地址,也是相对地址,我们将这些地址信息称为 IAT(Import Address Table,导入地址表),而 OriginalFirstThunk 仍然指向原来的 IMAGE_THUNK_DATA 数组,数组的内容也没变化,如图 12-22 所示。

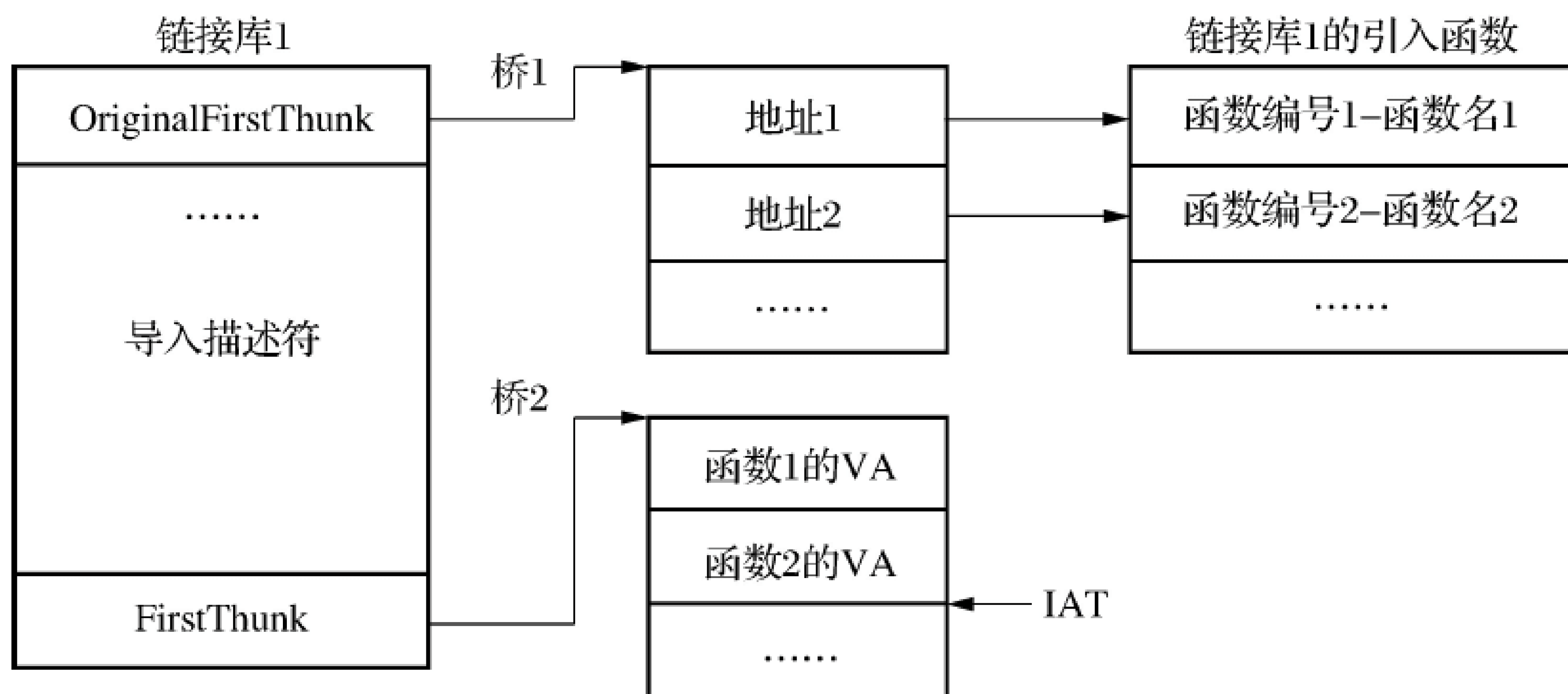


图 12-22 加载 DLL 之后 IMAGE_THUNK_DATA 数组的指向

3. 导入地址表

导入地址表(IAT)是一个内存中而非磁盘文件中的概念,也就是说,只有 PE 文件加载到内存中时,导入地址表才会生成。图 12-23 是采用 OD 加载了一个 GUI 的 EXE 文件后的结果,这实质上就是导入地址表。从图中可以看出,我们调用的是微软的运行时库 msvcrt.dll,一共调用了 6 个函数,这些函数的地址存放在一片起始地址为 0x00416520,结束地址为 0x00416538(0x00416534 +4)的内存区域中,每 4 字节的空间保存了一个被调函数的入口基址。

导入地址表(IAT)也是挂钩技术的重要实施对象,这是因为 IAT 中存放了导入函数的真

实地址,在这里进行拦截是一个非常好的选择,后续章节中会对这种技术进行描述。

```
00414444  .  C3              retn
00414445  >  E9 C6010000     jmp <jmp.&MSVCRT.terminate>              Jump to msvcrt.terminat
0041444A  $- FF25 34654100   jmp dword ptr [<&MSVCRT.memset>]
00414450  .- FF25 30654100   jmp dword ptr [<&MSVCRT.strncmp>]
00414456  $- FF25 2C654100   jmp dword ptr [<&MSVCRT.memcpy>]
0041445C  $- FF25 28654100   jmp dword ptr [<&MSVCRT.strlen>]
00414462  .- FF25 24654100   jmp dword ptr [<&MSVCRT._ismbcdigit>]
00414468  .- FF25 20654100   jmp dword ptr [<&MSVCRT.free>]
0041446E  >$ 833D 54F04100 FF cmp dword ptr [41F054], -1
```

图 12－23　导入地址表在内存中的形态

4. 重定向表

重定向表是为了解决动态加载地址的修正问题而引入的表。我们在前文说过,一般 DLL 的默认基址是 0x10000000,EXE 文件的默认基址是 0x00400000。EXE 文件一般不存在地址重定向的问题,因为一个进程中只有一个 EXE 文件存在,且是最先加载运行的那个,没人跟它抢基址。但是 DLL 就不一样了,一个进程可能会用到多个 DLL 模块,每个模块的默认基址都是 0x10000000,这样就会产生基址争用的问题,先加载者得默认基址,后面的只能另选他处了。为了帮助不能在默认基址加载的 PE 模块在新的地址上正确修正变量地址,重定向表就被引入了。

重定向表的基本思想是:将 PE 文件中需要“写死”的变量地址识别出来,以 RVA(相对虚拟地址)来替代它们,之所以采用 RVA,是因为相对地址是个差值,是不以基址的变化而变化的偏移值,这就很好地迎合了默认基址多变的情况。当 PE 模块被加载到内存的时候,真实的加载基址直接加上事先记录的这个偏移值就可以了。

编号	名称	偏移	大小	数据	备注
1	VirtualAddress	004690DC	4	00002000	
2	SizeOfBlock	004690E0	4	0000001C	
3	Relocation	004690E4	2	3054	HighLow:00002054
4	Relocation	004690E6	2	3083	HighLow:00002083
5	Relocation	004690E8	2	30D6	HighLow:000020D6
6	Relocation	004690EA	2	3103	HighLow:00002103
7	Relocation	004690EC	2	314D	HighLow:0000214D
8	Relocation	004690EE	2	326C	HighLow:0000226C
9	Relocation	004690F0	2	3B15	HighLow:00002B15
10	Relocation	004690F2	2	3B1E	HighLow:00002B1E
11	Relocation	004690F4	2	3B31	HighLow:00002B31
12	Relocation	004690F6	2	0000	Absolute:00002000

图 12－24　重定向表结构视图

重定向表由一个虚拟地址(VirtualAddress)、一个块大小(SizeOfBlock)和一堆 Block 组成,如图 12－24 所示。一个重定向表涵盖了一个内存页面(4 KB 大小)中需要重定向的变量的地址的偏移(Offset)。例如 PE 文件的.text 区段占用了 2 个内存页面,第一个内存页面含有 10 个需要重定向的变量,第二个内存页面含有 5 个需要重定向的变量,基于此,第一个重定向表含有 10 个 Block,第一个内存页面的 VirtualAddress 为 0x10001000;第二个重定向表含有 5 个

Block，第二个内存页面的 VirtualAddress 为 0x10002000。

- **VirtualAddress**：占用 4 字节，表示该重定向表要表示的内存页面的基址。
- **SizeOfBlock**：占用 4 字节，表示有效的 Block 占用的字节数加上 VirtualAddress 和 SizeOfBlock 的大小。使用 SizeOfBlock 可以计算出它之后携带了多少 Block。
- **Block**：占用 2 字节，低 12 位表示需要修正的变量在该页面中的位移，高 4 位表示该 Block 是否是个有效块。

我们用一个图来形象地表示重定向表，如图 12－25 所示。

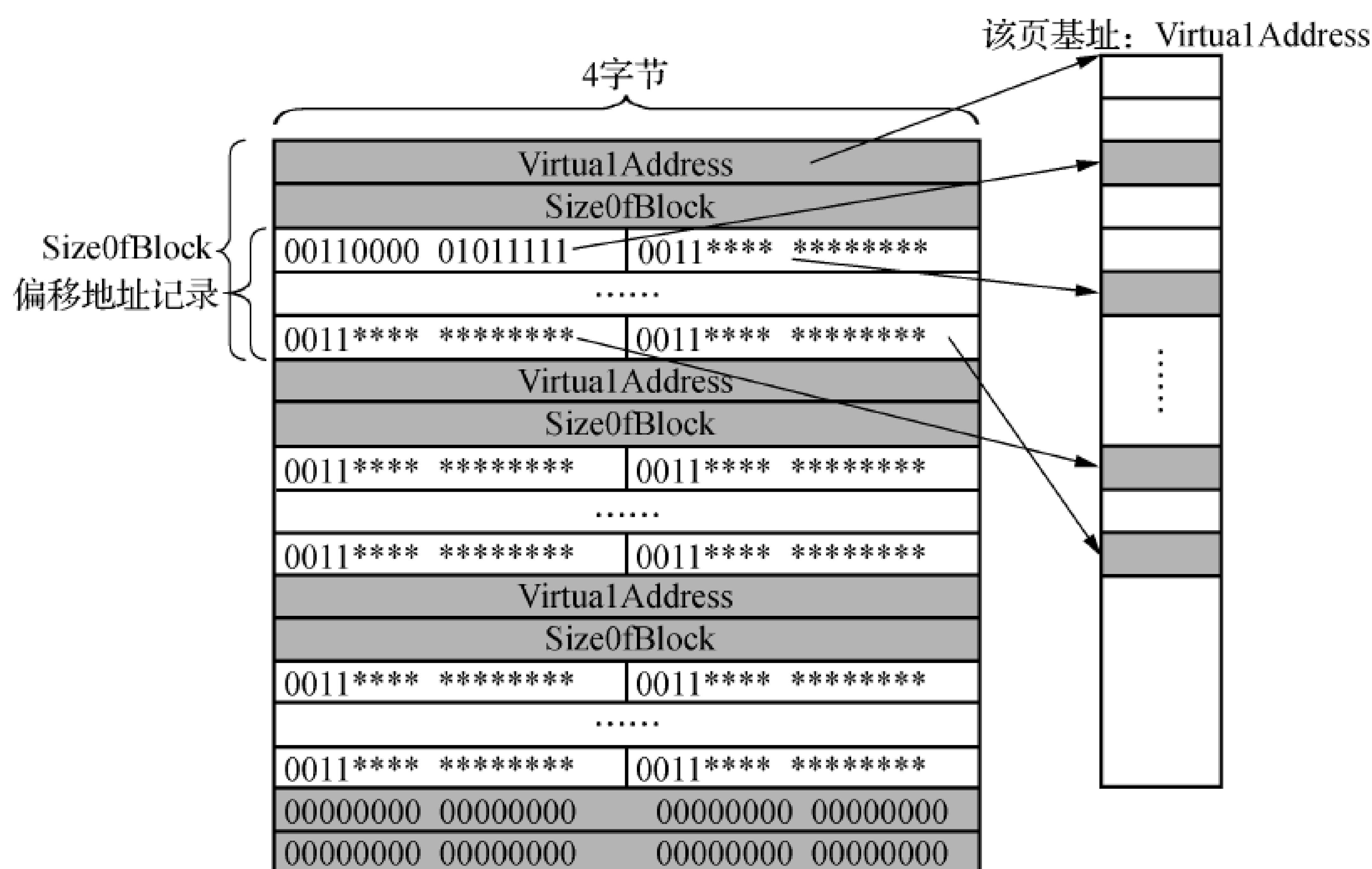


图 12－25　一个重定向表的实例

从图中可见，有 3 个内存页面需要重定向，各个重定向表都是紧挨着的，最后有一个全为 0 的 8 字节区域，表示重定向表的结束。

5. 延迟导入表

延迟导入表是用来描述动态链接库延迟加载信息的专用表。在加载和运行时，有些函数可以等到实际使用的时候再去加载对应的 DLL，没必要在程序一开始加载的时候就全部初始化好，采用延迟导入可以使初始的加载启动速度更快、使用的内存更少。延迟导入表的本质就是在 IAT 中放入固定的跳转代码，而不是加载时使用被调入函数的真实地址填充，待到真正调用的时候才重写 IAT 为真实的函数入口地址。

一般在编译器中，针对延迟导入会有相应的配置选项，例如 Visual C++ 6.0(VC 6)中的 Delay Loaded Dlls 选项中可以配置需要延迟导入的 DLL，其中每一个要延迟加载的 DLL 都对应一个延迟导入表。延迟导入表的数据结构为 ImgDelayDescr，如下所示：

```
typedef struct ImgDelayDescr {
    DWORD   grAttrs;            //1 表示后面的地址都是 RVA,0 表示后面的地址都是指针
    RVA     rvaDLLName;         //要导入 DLL 的名称的 RVA
    RVA     rvaHmod;            //要导入 DLL 的基址,导入前为空
    RVA     rvaIAT;             //IAT 的 RVA,导入前 IAT 中存放跳转代码,导入后为导入函数地址
    RVA     rvaINT;             //INT 的 RVA
```

```
    RVA     rvaBoundIAT;          //绑定导入表的 RVA
    RVA     rvaUnloadIAT;         //延迟导入卸载
    DWORD   dwTimeStamp;          //延迟导入 DLL 的时间戳
} ImgDelayDescr, * PImgDelayDescr;
```

前面说过延迟导入表的本质就是在 IAT 中放入了跳转代码,从而更改原先的加载流程,这个跳转代码如下所示:

```
.text:75C7A363 _imp_load_IntenetConnectA@ 32:    ; InternetConnectA(x,x,x,x,x,x,x,x)
.text:75C7A363                 mov     eax,offset _imp_InternetConnectA@ 32
.text:75C7A363                .jmp     _tailMerge_WININET
```

跳转代码将要跳转的函数的地址放入 EAX 寄存器中,之后跳转到_tailMerge_WININET。我们能够猜测到,EAX 寄存器的值是作为_tailMerge_WININET 的参数的。当第一次调用 DLL 的某函数时,会首先跳转到上述地址执行 EAX 寄存器赋值,然后再跳转到_tailMerge_WININET。

_tailMerge_WININET 的代码如图 12 – 26 所示,首先保存若干寄存器的值和延迟导入表,再调用_delayLoadHelper,最后跳转到 EAX 寄存器存放的地址处(_delayLoadHelper 会对 EAX 寄存器赋新值)。_delayLoadHelper 负责加载 DLL,查找导出函数并填充至 IAT 中,执行完成后 IAT 中已存在了函数地址,并将该地址赋值给 EAX 寄存器。当跳转到 EAX 也就相当于跳转到真实函数地址了。如此便完成了延迟导入表的初始化和使用,后面如果再调用相同函数的话就跟访问 IAT 一样,不会再走_delayLoadHelper 了。延迟导入表的执行流程如图 12 – 27 所示。

其中，_DELAY_IMPORT_DESCRIPTOR_WININET就是ImgDealyDescr

```
__tailMerge_WININET proc near
.text:75C6BEF0                  push    ecx
.text:75C6BEF1                  push    edx
.text:75C6BEF2                  push    eax
.text:75C6BEF3                  push    offset __DELAY_IMPORT_DESCRIPTOR_WININET
.text:75C6BEF8                  call    __delayLoadHelper
.text:75C6BEFD                  pop     edx
.text:75C6BEFE                  pop     ecx
.text:75C6BEFF                  jmp     eax
.text:75C6BEFF __tailMerge_WININET endp
```

图 12 – 26　_tailMerge_WININET 的代码

6. 资源表

资源表在 GUI 进程的 EXE 文件中非常常见。本节我们以一个 GUI 进程的 EXE 文件为例来讲解资源表。

从图 12 – 28 中我们可以看出,该 EXE 文件包含了 Icons、Dialogs、Icon Groups 和 Version Info 四部分资源。资源表结构体有很多种,包括根目录类型(Resource Directory,资源目录头)、子目录类型(Resource Directory Entry,资源目录项)、文件类型(Resource Data Entry,资源数据)等,常用的也是这三种。图 12 – 28 展示了这三种类型之间的关系,图 12 – 29 则展示了从资源目录头找到资源数据的过程。

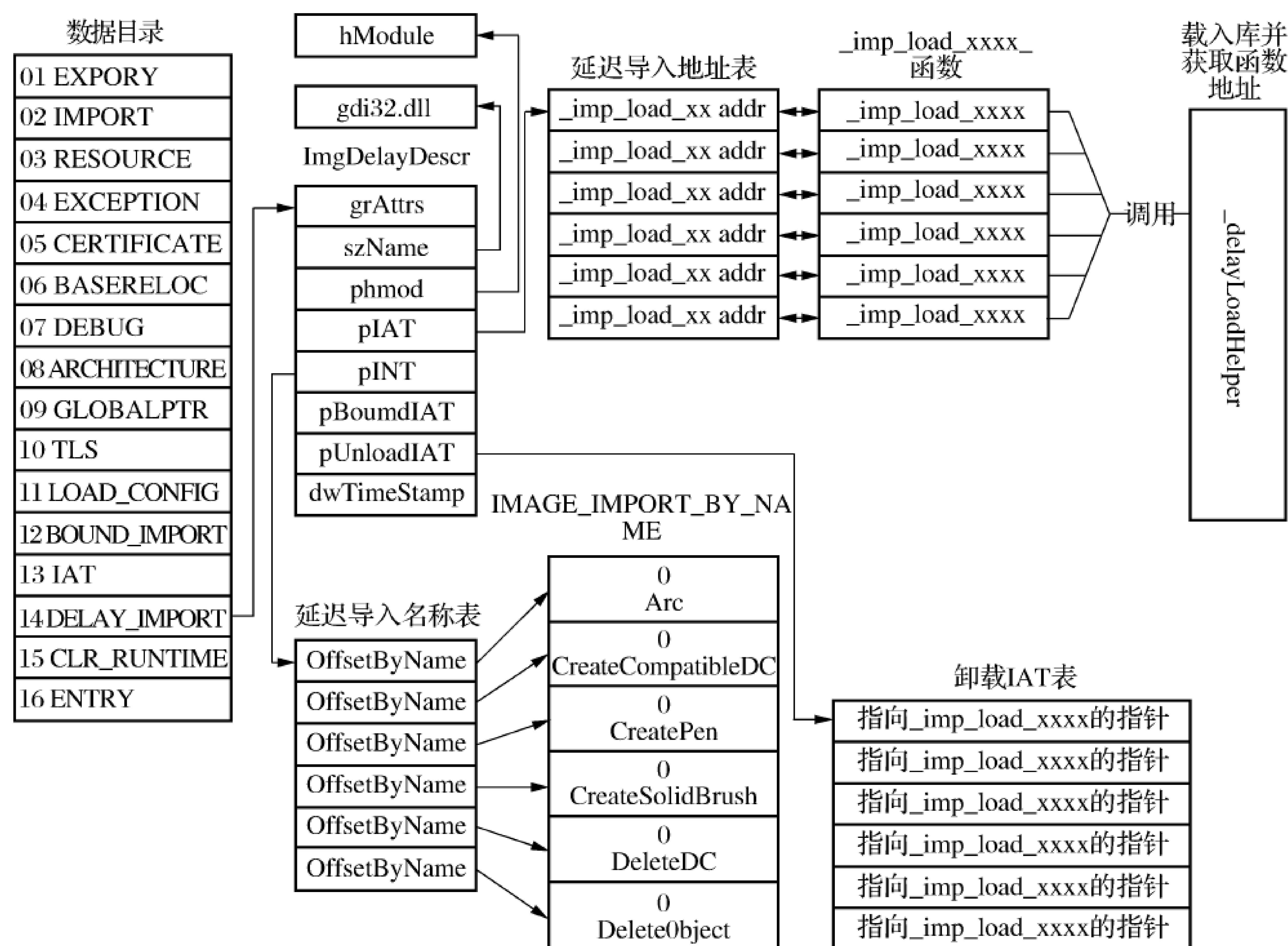
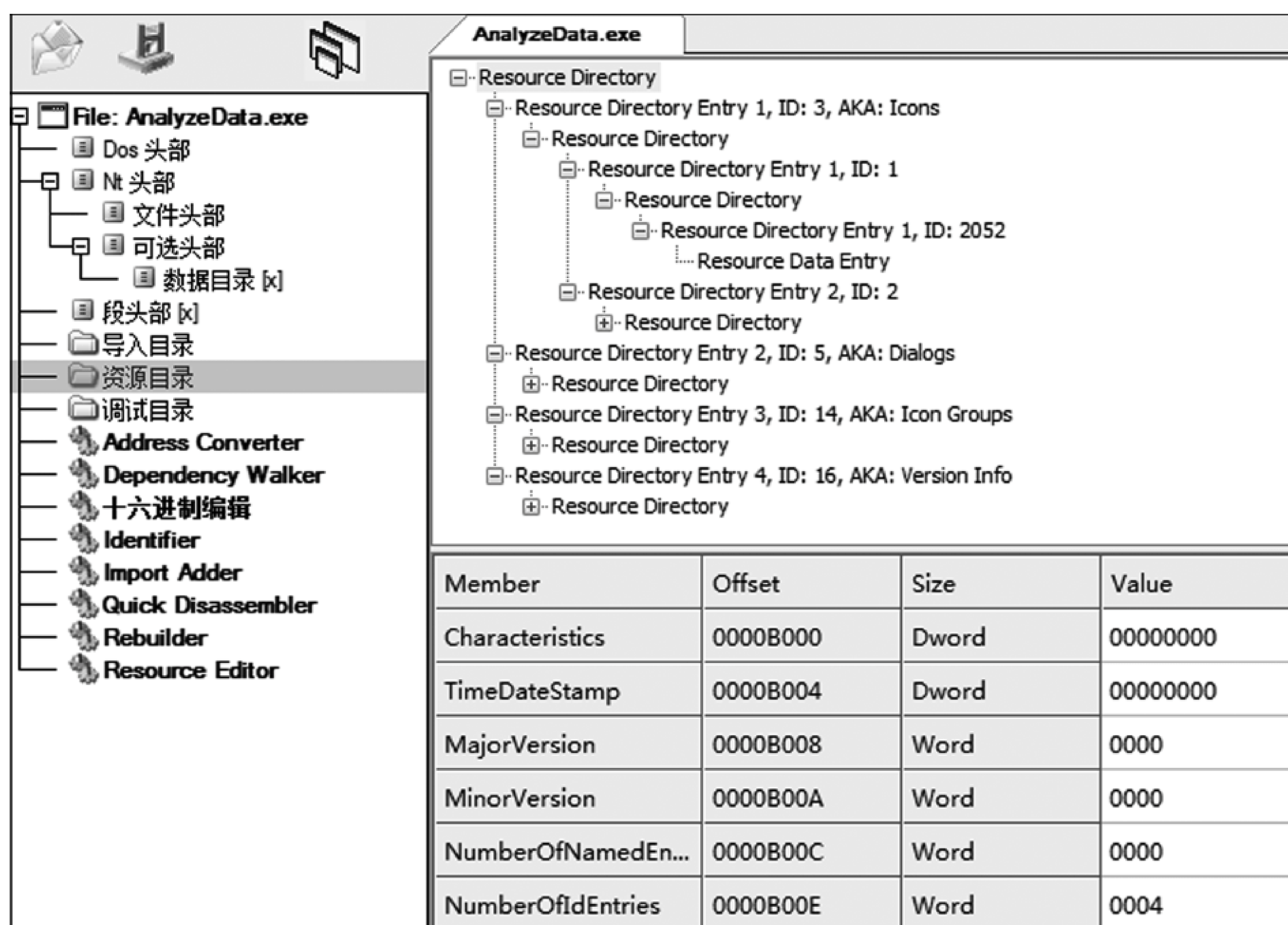


图 12－27　延迟导入表的执行流程



Member	Offset	Size	Value
Characteristics	0000B000	Dword	00000000
TimeDateStamp	0000B004	Dword	00000000
MajorVersion	0000B008	Word	0000
MinorVersion	0000B00A	Word	0000
NumberOfNamedEn...	0000B00C	Word	0000
NumberOfIdEntries	0000B00E	Word	0004

图 12－28　资源表结构视图

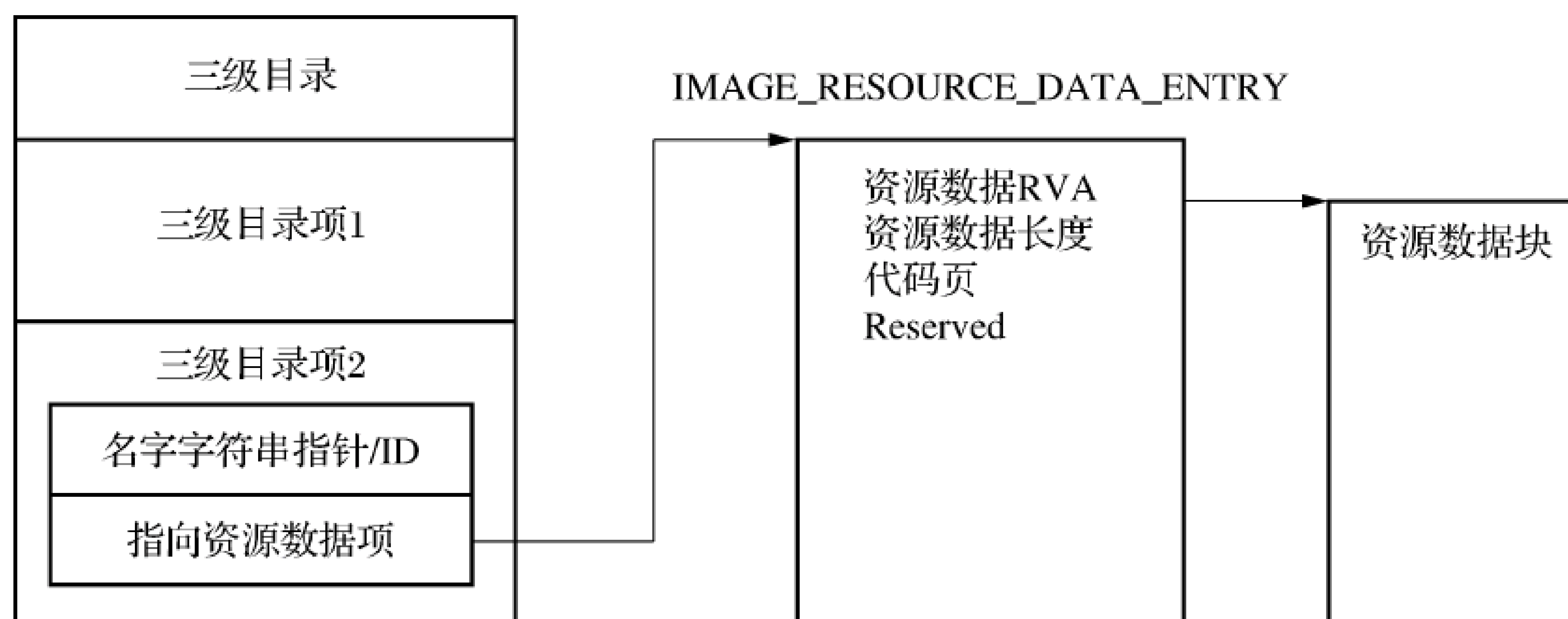


图 12－29　从资源目录头找到资源数据

（1）资源目录头表的定义如下：

```
typedef struct_IMAGE_RESOURCE_DIRECTORY {
    DWORD   Characteristics;          //资源属性,总为 0
    DWORD   TimeDateStamp;            //时间戳
    WORD    MajorVersion;             //资源大版本号,总为 0
    WORD    MinorVersion;             //资源小版本号,总为 0
    WORD    NumberOfNamedEntries;     //按照名称命名的数量,即按照字符串加载的资源数量
    WORD    NumberOfIdEntries;        //按照 ID 命名的数量,即按照 ID 加载的资源数量,最常用
//  IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[];
} IMAGE_RESOURCE_DIRECTORY, * PIMAGE_RESOURCE_DIRECTORY;
```

（2）资源目录项表的定义如下：

```
typedef struct_IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
       struct {
          DWORD  NameOffset:31;            //低 31 位为偏移,定义了目录项的名称或者 ID
          DWORD  NameIsString:1;           //若为 1,则低 31 位的偏移指向 Unicode 字符串的指针
       };
       DWORD  Name;
       WORD   Id;
    };
    union {
       DWORD  OffsetToData;
       struct {
          DWORD  OffsetToDirectory:31;     //若高位为 1,则 RVA 偏移指向的是新的(根目录)
          DWORD  DataIsDirectory:1;
       };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY, * PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

（3）资源数据项表的定义如下：

```
typedef struct_IMAGE_RESOURCE_DATA_ENTRY {
    DWORD   OffsetToData;          //资源数据的偏移 RVA
    DWORD   Size;                  //大小
    DWORD   CodePage;              //代码页缓冲,未使用
    DWORD   Reserved;              //未使用
} IMAGE_RESOURCE_DATA_ENTRY, * PIMAGE_RESOURCE_DATA_ENTRY;
```

因为资源是有父子关系的,所以 Windows PE 文件按照资源树的方式组织各种资源,而资源表可以将资源的属性、名称、父子关系等信息串联起来,构筑资源视图,如图 12－30 所示。我们使用 PE 工具分析上述 EXE 文件,将每种资源的详细信息一目了然地展示了出来。





图 12-30　GUI EXE 模块的资源视图

12.2.2.4　区段表与区段

区段表在 PE 文件中的作用也很重要,它表明了代码段、数据段、资源段等区段的位置、大小、属性等重要信息。每个区段都是内存页面(4KB)对齐的,PE 加载器使用区段表将区段对齐并映射到内存页面中。我们先来看区段表的结构,如图 12-31 所示。

编号	名称	偏移	大小	数据	备注
1	Name	000001F8	8	.text	.text
2	VirtualSize	00000200	4	002DEEF2	
3	VirtualAddress	00000204	4	00001000	
4	SizeOfRawData	00000208	4	002DF000	
5	PointerToRawData	0000020C	4	00001000	
6	PointerToRelocations	00000210	4	00000000	
7	PointerToLinenumbers	00000214	4	00000000	
8	NumberOfRelocations	00000218	2	0000	
9	NumberOfLinenumbers	0000021A	2	0000	
10	Characteristics	0000021C	4	60000020	

图 12-31　区段表结构视图

区段表在 WinNT. h 中也有定义，即 IMAGE_SECTION_HEADER 结构。每个区段都对应一个区段表，图 12 – 31 表明一共有 5 个区段，因此也就对应了 5 个区段表。每个区段表的具体域值含义如下所示：

```
typedef struct_IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME]; //该区段的名字，例如.text
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;            //该区段在内存中的大小
    } Misc;
    DWORD   VirtualAddress;                 //该区段在内存中的相对虚拟地址
    DWORD   SizeOfRawData;                  //该区段在磁盘文件中的大小，磁盘页面对齐
    DWORD   PointerToRawData;               //该区段在磁盘文件中的位置
    DWORD   PointerToRelocations;           //COFF 文件中使用，一般为 0
    DWORD   PointerToLinenumbers;           //COFF 文件中使用，一般为 0
    WORD    NumberOfRelocations;            //COFF 文件中使用，一般为 0
    WORD    NumberOfLinenumbers;            //COFF 文件中使用，一般为 0
    DWORD   Characteristics;                //区段属性，与 COFF 文件区段属性的取值范围相同
} IMAGE_SECTION_HEADER, * PIMAGE_SECTION_HEADER;
```

PE 文件在内存中与在磁盘中的布局是不一样的。每个区段要占用完整的页面（包括内存页面和磁盘页面），因此看上去会有缝隙（用 0 填充），这主要是因为内存和磁盘页面对齐的粒度不一样，即 SectionAligment 与 FileAligment 不相等，代码或数据也不可能正好占据一整个页面，而且内存页面的颗粒度大于磁盘页面，这就造成了如图 12 – 32 所示的效果。

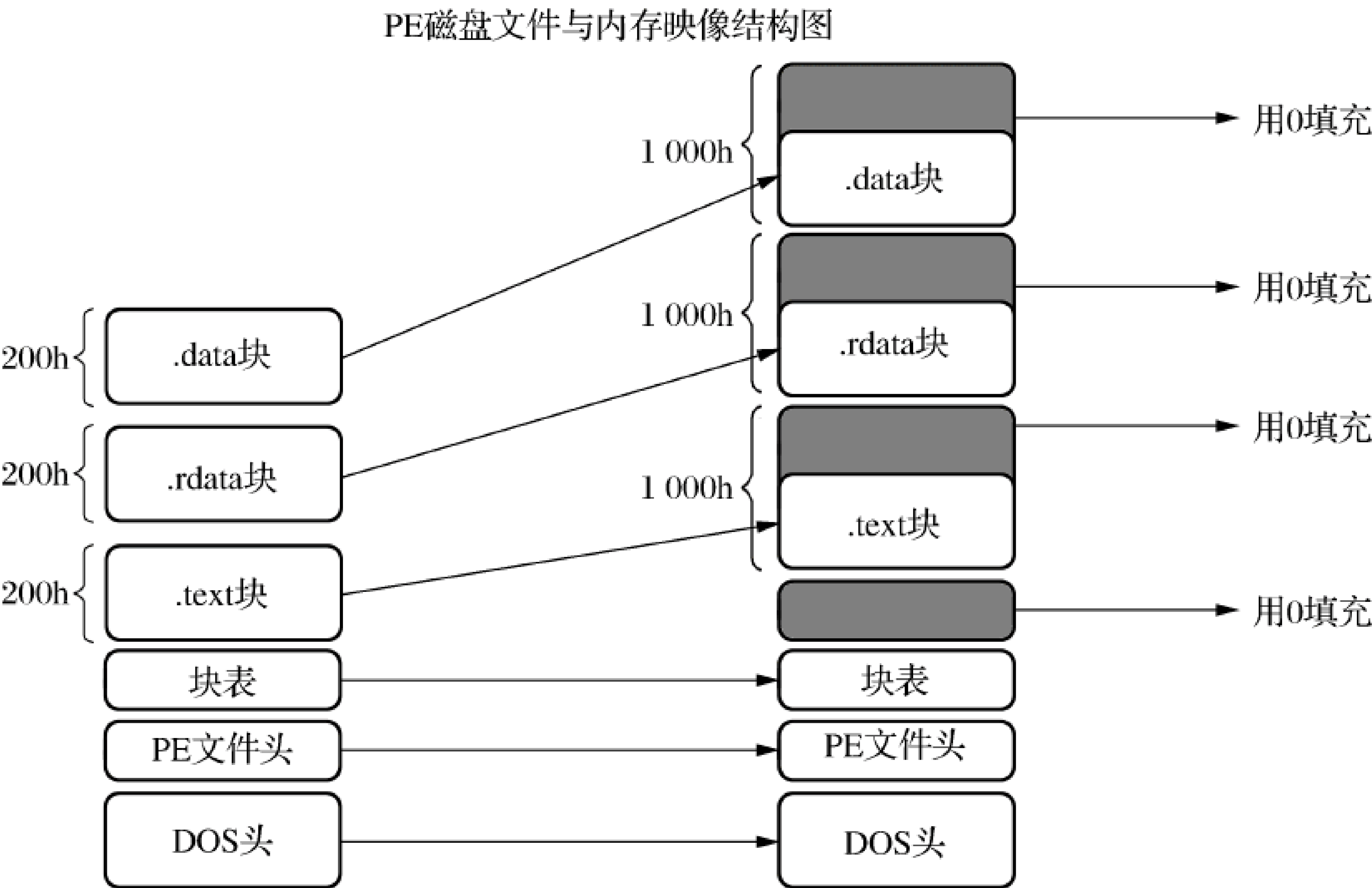


图 12 – 32　PE 文件在内存中与磁盘中的布局视图

不过现在对齐粒度逐渐趋于统一，对某些系统来说，SectionAligment 与 FileAligment 都是 0x1000，即 4KB，这样 PE 文件在内存与磁盘中的布局就统一起来了。一般来说，DOS 头、DOS 存根程序、NT 头以及区段表等描述性数据结构在内存中占据一个页面，各区段分别占用整数倍的页面。

PE 文件将数据表作为非描述性内容存放在各区段中。例如在图 12 – 33 中，. rdata 区段




中就包含了导出表、导入表、调试表、导入地址表,资源表在. rsrc 区段(资源段)中,重定向表在. reloc 区段(重定向段)中,而代码的入口当然要在. text 区段(代码段)中了。

编号	名称	偏移	大小	数值	备注
1	.text	00001000	1	00001000	0x1000 to 0x2DFEF2
2	Code Entry Point	00000128	4	00294AF1	

编号	名称	偏移	大小	数值	备注
1	.rdata	002E0000	1	002E0000	0x2E0000 to 0x33B1D8
2	Export Directory	0033B160	4	0033B160	
3	Import Directory	00333BF8	4	00333BF8	
4	Debug Directory	002E0B30	4	002E0B30	
5	Import Address Table	002E0000	4	002E0000	

编号	名称	偏移	大小	数值	备注
1	.data	0033C000	1	0033C000	0x33C000 to 0x46AC5C

编号	名称	偏移	大小	数值	备注
1	.rsrc	00468000	1	0046B000	0x46B000 to 0x46B390
2	Resource Directory	00468000	4	0046B000	

编号	名称	偏移	大小	数值	备注
1	.reloc	00469000	1	0046C000	0x46C000 to 0x4B3286
2	Base Relocation Table	00469000	4	0046C000	

图 12-33　各区段中包含的重要信息表

表 12-3　PE 文件中的区段

区段类型	区段释义	区段含义
. text	代码段	内容是指令代码,链接器会把. obj 文件中的. text 区段连接成一个大的. text 区段
. data	普通数据段	全局变量、静态变量存放于该数据段
. rdata	只读数据段	一般用来存放说明字符串和不可修改的数据
. idata	导入数据段	编译器已将该区段废弃,内容合并到了只读数据段
. edata	导出数据段	编译器已将该区段废弃,内容合并到了只读数据段
. rsrc	资源段	包括模块的全部资源,如图标、菜单、位图等存放于该区段,这是个只读数据段
. bss	未初始化数据段	编译器已将该区段废弃,内容合并到了普通数据段
. crt	运行时段	用于 C++ 运行时(CRT)所添加的只读数据,比较少见
. tls	线程局部存储段	存储当前线程的上下文信息,用于线程变量访问,与全局变量相比,不会影响其他线程
. reloc	重定向段	存放重定向表等数据
. pdata	异常信息段	存放异常信息数据

Windows PE 文件依靠这些头部数据结构和区段,将 EXE、DLL、SYS 等文件组织起来,形成内存中的进程。这也是我们开发应用程序时最常见到的编译链接后生成的文件形态。

最后要强调一点,除了上述 Windows 定义的区段,PE 文件的开发者还可以自己定义区段以及区段的使用机制。例如 TCPIP 协议栈驱动的 tcpip. sys 模块就有若干自己定义的区段,如 PAGE、PAGEIDP、PAGEIPSE、PAGECONS、INIT 这 5 个区段,如图 12 - 34 所示。自己来定义区段以满足特殊用途也是 Rootkit 技术中的常用手段。

tcpip.sys

Name	Virtual Size	Virtual Addr...	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
00000338	00000340	00000344	00000348	0000034C	00000350	00000354	00000358	0000035A	0000035C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0013D2E6	00001000	0013D400	00000400	00000000	00000000	0000	0000	68000020
.rdata	00024CA0	0013F000	00024E00	0013D800	00000000	00000000	0000	0000	48000040
.data	00037860	00164000	0000D200	00162600	00000000	00000000	0000	0000	C8000040
.pdata	00014DFC	0019C000	00014E00	0016F800	00000000	00000000	0000	0000	48000040
PAGE	00002B42	001B1000	00002C00	00184600	00000000	00000000	0000	0000	60000020
PAGEIDP	00002DCE	001B4000	00002E00	00187200	00000000	00000000	0000	0000	60000020
PAGEIPSE	0001BAC3	001B7000	0001BC00	0018A000	00000000	00000000	0000	0000	60000020
PAGECONS	000000F0	001D3000	00000200	001A5C00	00000000	00000000	0000	0000	40000040
INIT	0000563A	001D4000	00005800	001A5E00	00000000	00000000	0000	0000	E2000020
.rsrc	00020E10	001DA000	00021000	001AB600	00000000	00000000	0000	0000	42000040
.reloc	00000A68	001FB000	00000C00	001CC600	00000000	00000000	0000	0000	42000040

图 12 - 34 tcpip. sys 自定义的区段

本章小结

本章介绍了 Windows 可移植执行文件(PE 文件)的相关概念,包括 PE 头、DOS 头、NT 头等重要头部结构,以及导入表、导出表、导入地址表、重定向表、延迟导入表、资源表等重要的表结构,最后介绍了区段表和所表示的区段。

PE 结构是 Windows 进程编译、链接和加载的基础,也是逆向工程、恶意植入、反入侵等技术的核心。

第13章　Windows 启动过程

Windows 的启动是个漫长且参与者众多的过程。在 Windows 启动之前,无论是什么操作系统,都会经历计算机通电自检以及自检完成后的启动过程,这个自检后的启动过程又包括预引导、引导、内核载入、初始化及系统登录五大步骤。而 Windows 启动前后的参与者还包括 ROM 中的 POST 代码、BIOS/EFI/UEFI、MBR、引导扇区、NTLDR、操作系统内核组件 NTOSKRNL/HAL/BOOTVID/KDCOM,以及系统服务进程 smss. exe、csrss. exe、winlogon. exe 等等。

本章我们将按照图 13－1 所示的提纲介绍系统启动过程。

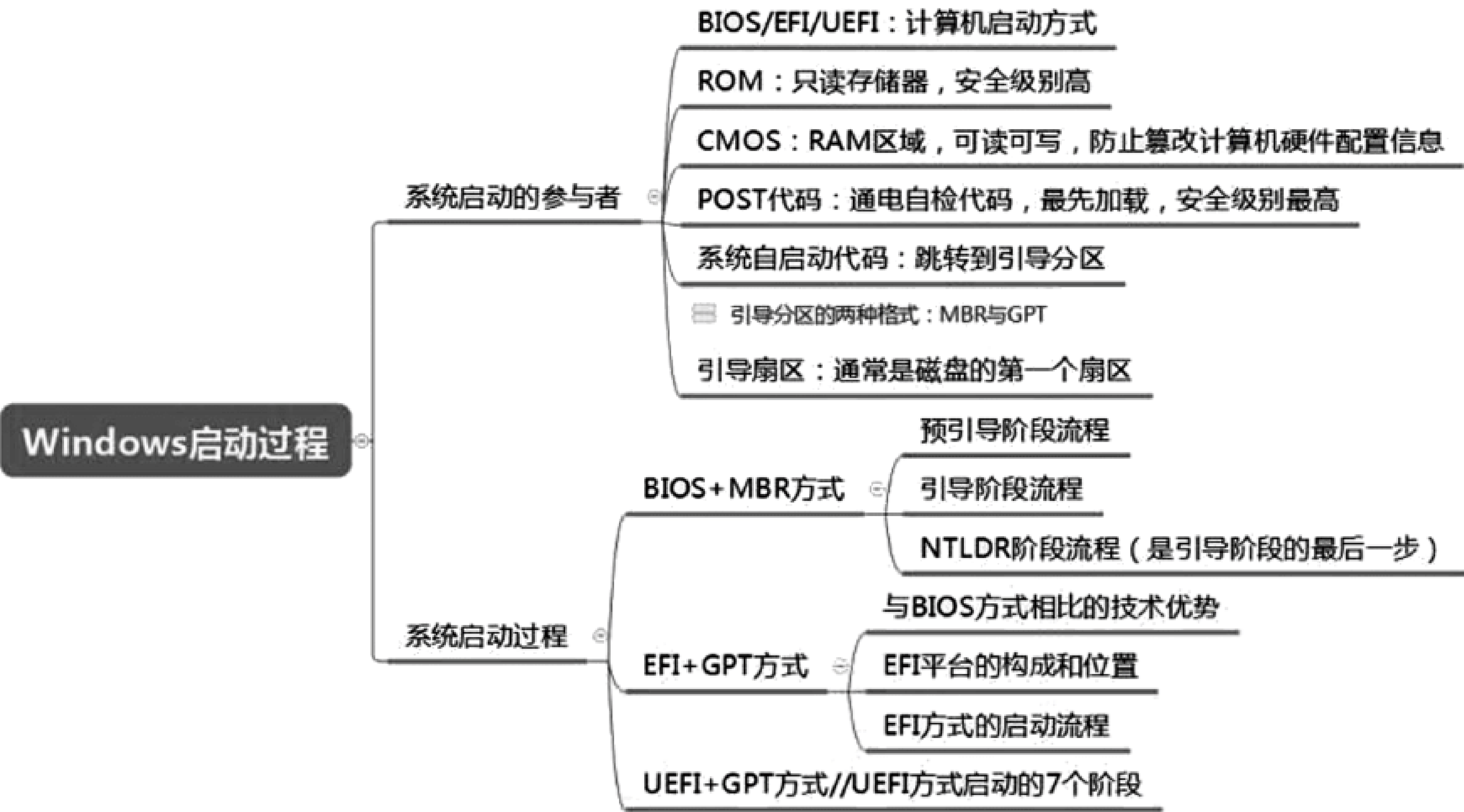


图 13－1　本章提纲

13.1　系统启动的参与者

在讲述启动过程之前,我们先来看一些重要的术语。

1）BIOS

BIOS(Basic Input Output System),顾名思义就是基本输入输出系统。其实 BIOS 也是固化到 ROM 中的代码,保存着计算机加电自检(POST)代码、系统自启动代码、CMOS 设置代码等。我们常说"BIOS 芯片",其实就是表示这段代码存放在 ROM 芯片上而已。

BIOS 代码是一些汇编代码,在 16 位的实模式下调用 int 0x13 中断执行。16 位实模式能直接访问的内存为 1 MB(地址线为 20 位),因此 BIOS 也只能访问这 1 MB 的内存,其中前面 640 KB 是基本内存,是为 MS-DOS 保留的;后面 384 KB 作为扩展内存,是供开机所需硬件以

及其他各类 BIOS 使用的。

但随着技术的发展，目前市面上的计算机基本上采用 EFI/UEFI 而摒弃了 BIOS。

2）EFI

EFI（Extensible Firmware Interface），即可扩展固件接口，是由 Intel 在 2001 年为全新类型的固件体系结构、接口和服务提出的建议性标准。EFI 的作用与 BIOS 对等，也是在启动过程中完成硬件初始化。但 EFI 却是采用 C 语言编写的，并且摒弃了 BIOS 中的中断方式而改为采用 EFI 驱动加载的方式识别和初始化硬件。不过这个驱动也不是直接面向 CPU 的，因为它无法被 CPU 直接执行，而是由专用于 EFI 虚拟机上的指令支持的，需要在 EFI 驱动运行环境 DXE 上解释执行。与 BIOS 相比 EFI 是 32 位或 64 位的，因此可实现更大寻址。

3）UEFI

UEFI 是 EFI 的 2.0 版本，U（Unified）表示统一的，UEFI 就是指统一的可扩展固件接口。与 EFI 相比 UEFI 有了不少的改进：

➢ 具有完整的图形驱动功能，也兼容了 USB 接口的鼠标和键盘。
➢ 支持安全启动，CPU 硬件支持嵌入一个 TPM 芯片（可信计算中的可信平台模块），支持 UEFI 安全启动的主板会根据这个芯片内记录的硬件签名对硬件进行度量验证，只有符合认证签名的硬件驱动才会被加载。Windows 8 及以上的版本在操作系统加载过程中还要对硬件驱动检查签名，不过这就是操作系统级别的安全机制了。

总结下来，无论是 EFI 还是 UEFI，都有预加载环境、驱动执行环境和驱动程序这些必要部分，并且两者都仅支持 GPT（GUID 分区表）磁盘引导系统。UEFI 只支持 64 位操作系统。

4）ROM

ROM（Read-Only Memory）是只读存储器，是固化在主板上的存储器，例如 BIOS 就是存储在 ROM 上的，无法被重写。因此在具有可信计算芯片的系统中，ROM 中的代码往往作为可信计算根存在。

5）POST 代码

POST（Power On Self Test）是通电自检代码，计算机加电后要进行系统自检，看看处理器、内存、硬盘等固件是否可用，甚至是否有缺失。这部分代码是系统中最先加载的代码，安全级别最高，因此必须放在不可重写的 ROM 上。

6）系统自启动代码

系统自启动代码执行跳转操作跳到操作系统引导设备的分区（例如硬盘、启动光盘等），并将引导程序加载到内存从而开启操作系统的启动过程。

7）CMOS

CMOS（Complementary Metal Oxide Semiconductor），即互补金属氧化物半导体，现在 CMOS 泛指采用该技术生产的芯片。

计算机中的 CMOS 芯片是一块用来保存数据的 RAM，由主板上的锂电池供电。RAM 中的数据是可读写可更改的，因此需要发布出接口对其进行更改和读写，这个接口就是 CMOS





设置代码。

CMOS 本身存放了计算机硬件配置等信息，可以对其进行更改。这些信息虽然是放在 RAM 上的，但 CMOS 设置接口代码却放置在 BIOS 上。

8）MBR 与 GPT

MBR（Master Boot Record，主引导记录）和 GPT（GUID Partition Table，全局唯一标识分区表）是两种不同的磁盘分区类型，两者的主要区别在于不同的分区结构和分区方法。

MBR 支持的最大卷为 2 TB，超过 2 TB 则不能识别，并且每个磁盘最多有 4 个主分区，或 3 个主分区加 1 个扩展分区和无限制的逻辑驱动器。

GPT 支持的最大卷为 18 EB，且每个磁盘支持最多 128 个分区，同时 GPT 磁盘有多余的备份分区表来提高分区数据结构的完整性。目前的主流计算机都已支持了 GPT 磁盘。

9）引导扇区

引导扇区通常是磁盘的第一个扇区，一般是硬盘的 0 号柱面的 0 号磁头的 0 号扇区，用以加载操作系统，每个扇区的大小为 512 字节。

13.2 系统启动过程

介绍完了这些参与者，我们以 BIOS + MBR（BIOS 方式）、EFI + GPT（EFI 方式）和 UEFI + GPT（UEFI 方式）这三种方式来考察 Windows 系统的启动过程。先看最古老的 BIOS + MBR 的方式。

13.2.1 BIOS + MBR 方式

我们先来宏观地看一下 BIOS + MBR 方式的系统启动流程，如图 13－2 所示。

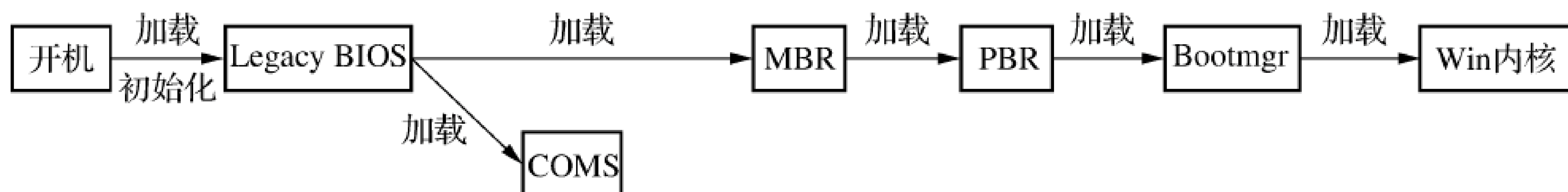


图 13－2 BIOS 方式的系统启动流程（图片来自参考文献）

BIOS 启动方式本身并不知道其他细节，只会执行在指定磁盘上的 MBR 上所发现的二进制启动代码。这种方式要经历预引导和引导两个阶段，图 13－3 所示的步骤是对 BIOS 方式的启动流程的描述。

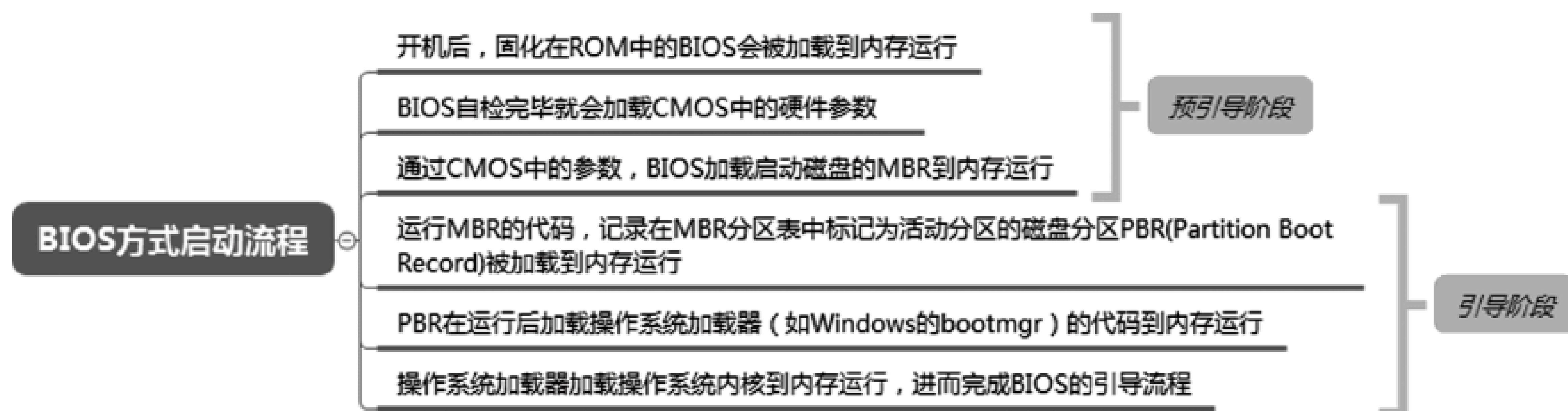


图 13－3 BIOS 方式的启动步骤的描述

在引导阶段的最后一步，要把操作系统内核加载到内存中运行，这一步是整个操作系统启动流程中最耗时的一步，而这一步也是我们常说的“NTLDR 阶段”，在这个阶段要经历以下若干步骤：

（1）使 CPU 从 16 位实模式进入 32 位保护模式，如图 13－4 所示。

➢ 实模式本质上是使用 16 位的寄存器去访问 20 位的地址线（1 MB 地址空间）而必须采用的段基址＋段偏移的内存地址访问机制。

➢ 保护模式本质上是以 32 位地址线访问 32 位地址空间的访问机制，即以段基址＋段偏移的逻辑地址访问 LDT/GDT 得到线性地址，用线性地址通过三级页面表得到物理地址的过程。

➢ 实模式的寻址过程无法全部利用系统中的内存，因此必须转化成保护模式。

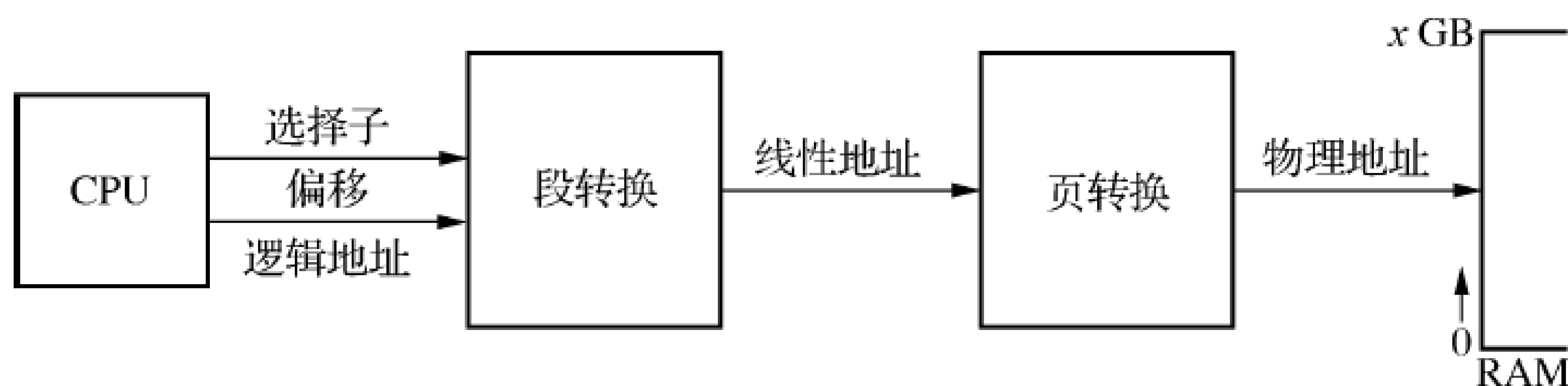


图 13－4　保护模式地址访问

（2）启用 CPU 的分页机制。

（3）判断：如果是 SCSI 硬盘，则加载 BtBootDD. sys 用于访问磁盘；否则，使用 int 0x13 指令将某扇区加载到内存。由于此时文件系统（FAT32 或 NTFS 等）尚未载入，因此需要有一个替代简易文件系统能识别磁盘上的分区，这就是加载 BtBootDD. sys 的用意。BtBootDD. sys 可以看作一个微型文件系统。

（4）如果发现有效的 hiberfil. sys（休眠管理），则加载并恢复 Hibernate。

（5）打开 boot. ini 文件，读取其中的设置，如果有多个选项，则显示菜单，用于选择要启动的操作系统，这主要是针对安装了多操作系统的情况。

（6）如果用户按过 F8 键，则显示选项菜单。

（7）加载并执行 ntdetect. com 进行硬件检测。ntdetect. com 调用 BIOS 收集系统的基本信息并形成一个表，表中内容包括时间、总线/适配器类型、磁盘信息、可移动磁盘信息、输入设备信息、浮点处理器信息、端口信息、串口信息、显卡信息等，并同时保存到注册表相应键下。

（8）显示启动进度条或启动 Splash 服务。

（9）加载系统目录下的 ntoskrnl. exe、hal. dll 及其依赖库（先载入 ntoskrnl. exe，后载入 hal. dll）。

（10）加载注册表的 System Hive，并加载其中定义 boot 类型的驱动程序。

（11）执行 ntoskrnl. exe 的入口函数，开启内核初始化过程。入口函数执行步骤包括：

① 执行 KeStartAllProcessors 初始化 CPU，这是内核初始化的起点，如图 13－5 所示。

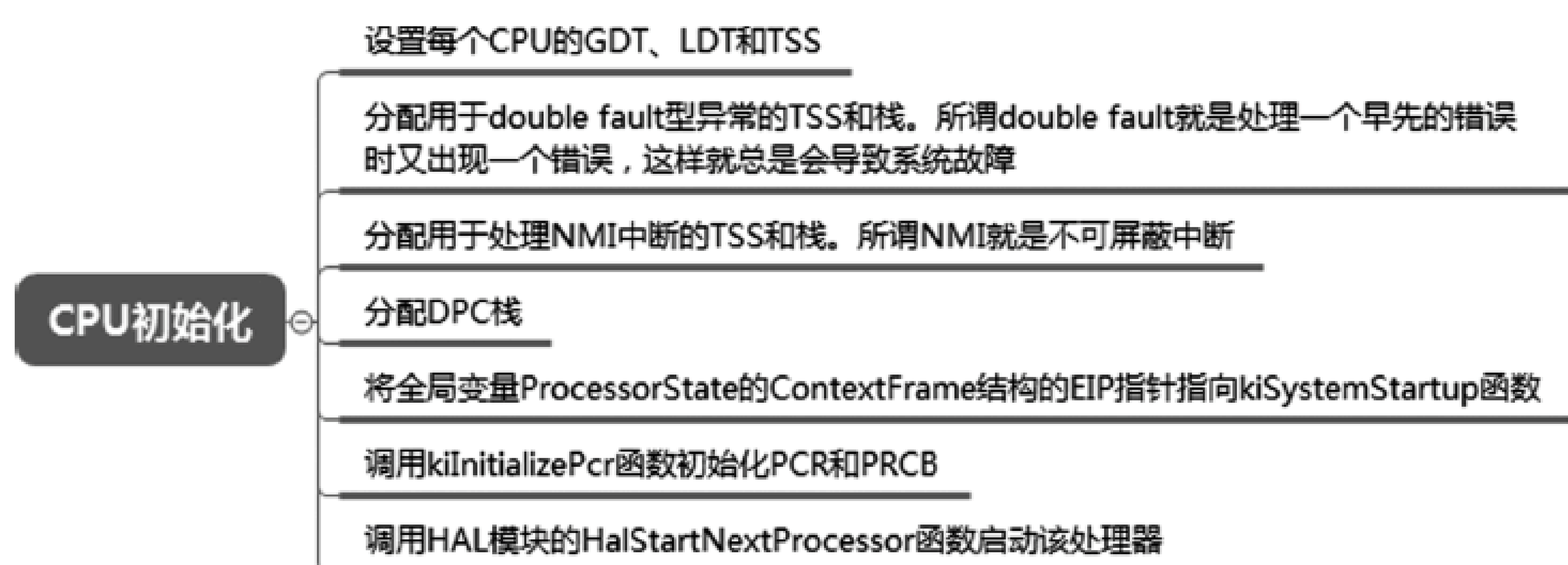


图 13－5　CPU 初始化过程

② 执行 KiSystemStartup 初始化一些系统硬件状态，并调用一些系统初始化过程，然后就进入调度程序，开始系统调度过程，如图 13－6 所示。

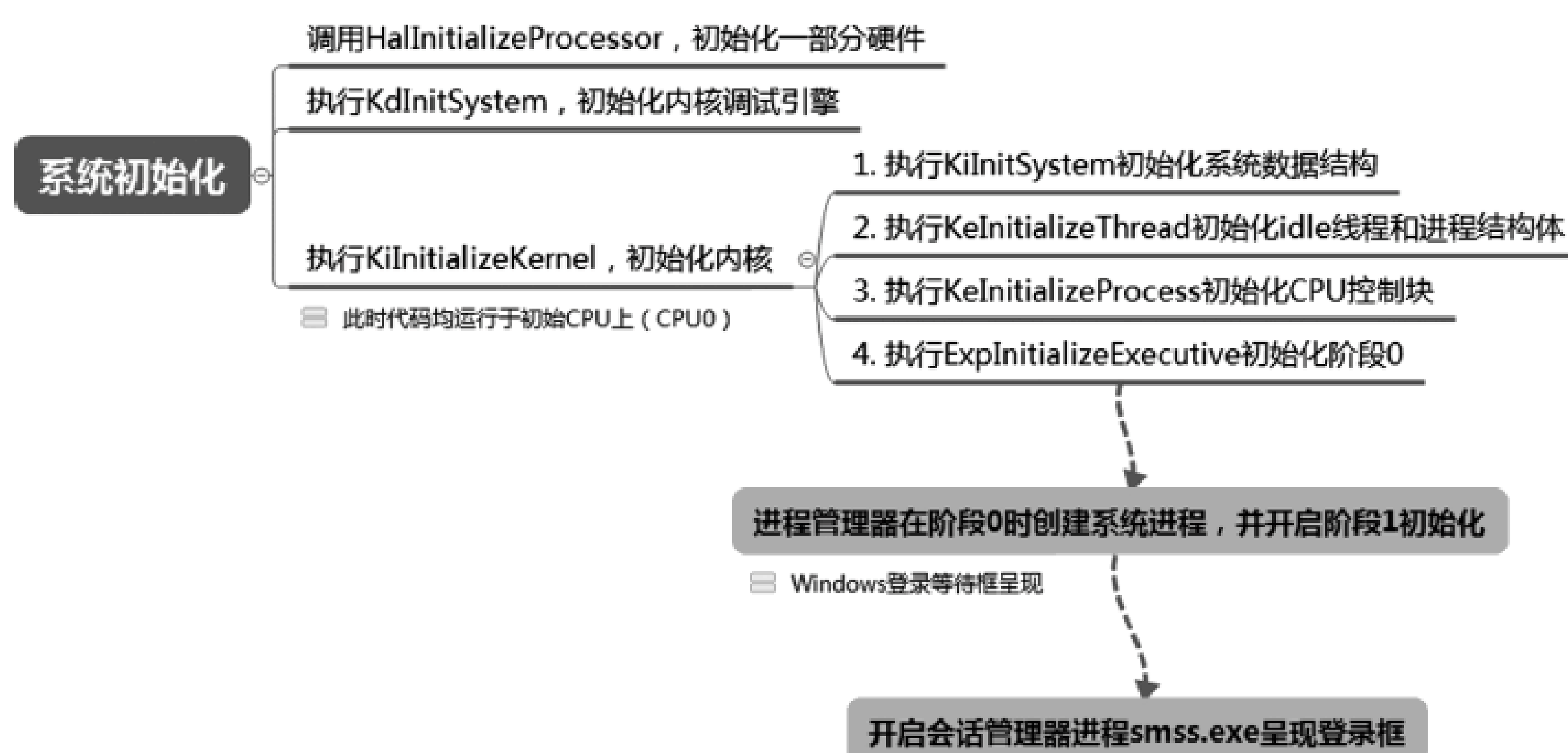


图 13－6　KiSystemStartup 的执行过程

这一步中还包含了初始化执行体，其详细过程如图 13－7 所示。

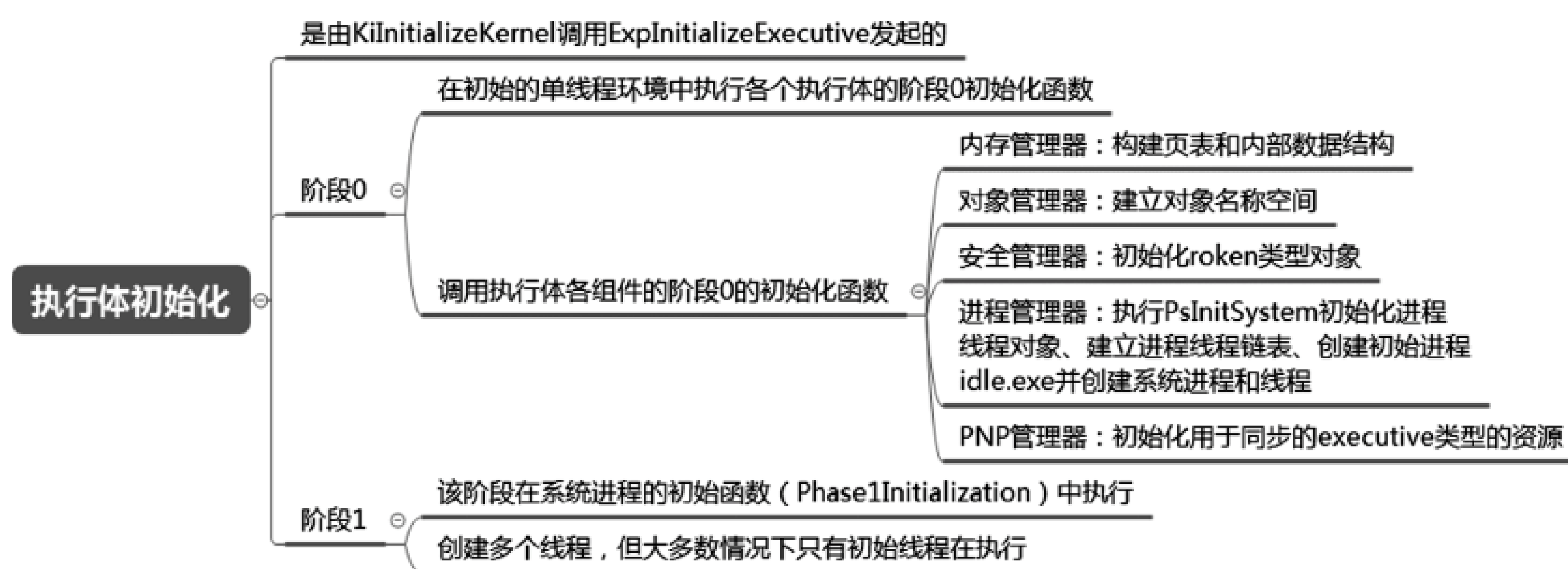


图 13－7　执行体初始化过程

当进入到执行体初始化这一阶段的时候，计算机屏幕上就会显示 Windows 版本的标志了，同时还会显示一个滚动的进度条。从这一步开始我们才能从屏幕上对系统的启动进度有一个直观的印象。这一阶段中主要会完成 4 项任务：

➢ 创建 Hardware 注册表键；

➢ 对 Control Set 注册表键进行复制；

➢ 载入和初始化设备驱动；

- 启动各个系统服务。

系统内核成功载入并且成功初始化所有底层设备驱动后，会话管理器开始启动高层子系统和服务，然后启动 Win32 子系统。Win32 子系统的作用是控制所有输入/输出设备以及访问显示设备。当所有这些操作都完成后，Windows 的图形界面就可以显示出来了，同时我们也可以使用键盘以及其他的 I/O 设备了。

接下来会话管理器启动 winlogon. exe 进程。至此，初始化内核阶段已经全部完成，用户可以登录系统了。

登录时，由会话管理器启动的 winlogon. exe 进程将会启动本地安全性授权（Local Security Authority）子系统（lsass. exe）。这一步之后，屏幕上将会显示 Windows 操作系统的欢迎界面或者登录界面。不过此时系统的启动还没有彻底完成，后台可能仍然在加载一些非关键的设备驱动或服务。这些驱动和服务加载完成后，操作系统启动过程才算全部完成。

13.2.2　EFI + GPT 方式

EFI 非常类似于一个微型操作系统，并且也具有操控所有硬件的能力，其作用包括以下两方面：

- 向操作系统的引导程序以及某些必须在计算机初始化时运行的应用程序提供一套标准的运行环境。
- 为操作系统提供一套与固件通信的交互协议接口。

作为为了全新类型的固件体系结构、接口和服务提出的建议性标准，EFI 的固件具有下面几项技术优势：

- 支持大容量磁盘（超过 2 TB）的引导能力。
- 更快的启动速度。
- 独立于 CPU 的体系结构（使用 EFI 虚拟机实现）。
- 不依赖于 CPU 的独立驱动程序。
- 灵活的预操作系统环境，包括网络功能。
- 模块化设计思想。

当然，EFI 毕竟只是硬件和预启动软件之间的接口规范，并不提供中断响应机制，同时也需要以解释的方式运行。

EFI 平台由如下几个部分组成：pre-EFI 初始化模块、EFI 驱动执行环境、EFI 驱动程序、EFI 系统装载器、EFI 上层应用和 GUID 磁盘分区。

其中，pre-EFI 初始化模块包括协议结构（负责与硬件直接交互），平台驱动，框架驱动（UEFI 扩展功能执行的基础，为 EFI 的执行提供全流程支撑）和兼容性支持模块（是 X86 平台 EFI 系统中的一个特殊模块，为不具备 EFI 引导能力的操作系统提供类似于 BIOS 的系统服务）。EFI 系统装载器负责引导 UEFI 本身或 Windows 等操作系统的启动。EFI 上层应用相当于扩展的应用程序。

EFI 在整个系统堆栈中的位置如图 13-8 所示,可见 EFI 是介于固件和操作系统之间的接口。图 13-9 描述了 EFI 方式的系统启动流程,图 13-10 则是对 EFI 方式启动步骤的描述。

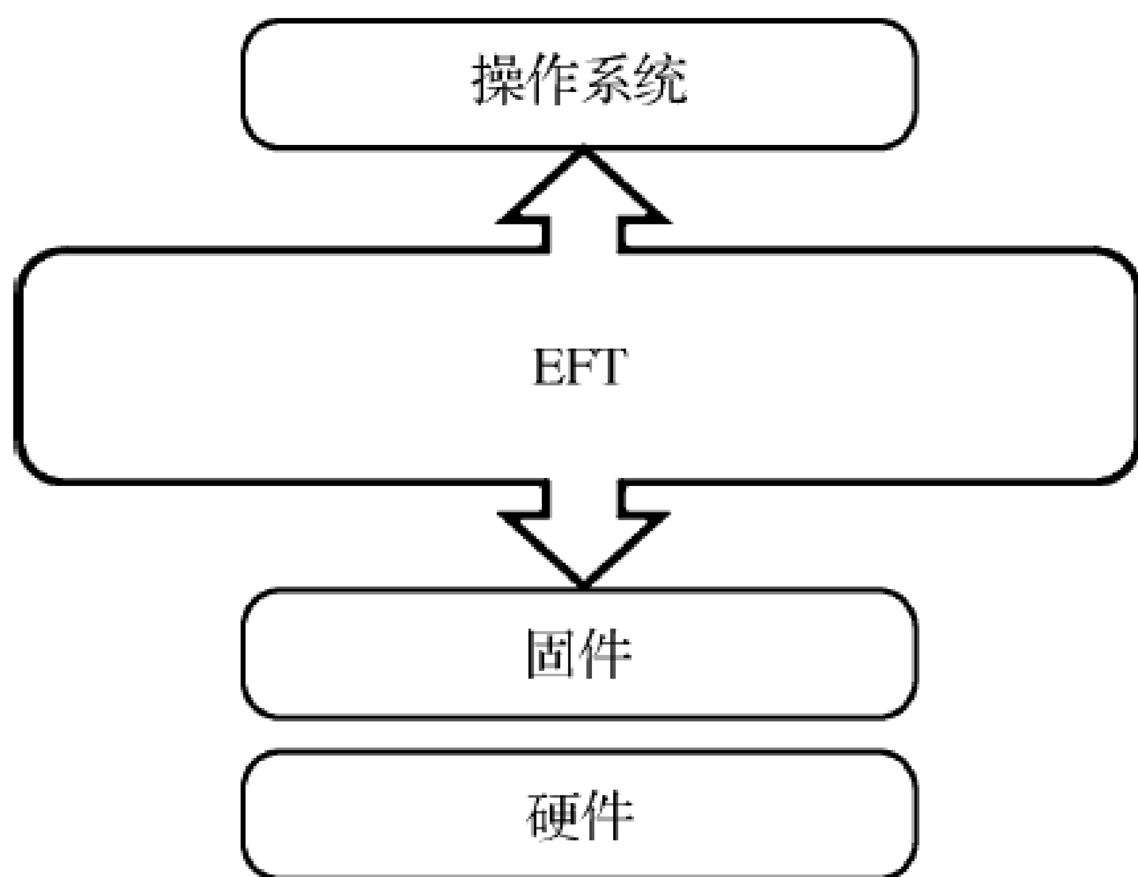


图 13-8　EFI 在整个系统堆栈中的位置

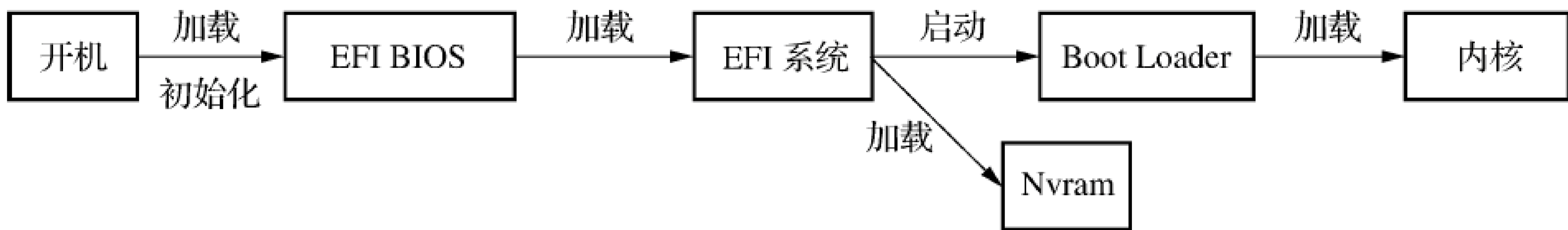


图 13-9　EFI 方式的系统启动流程

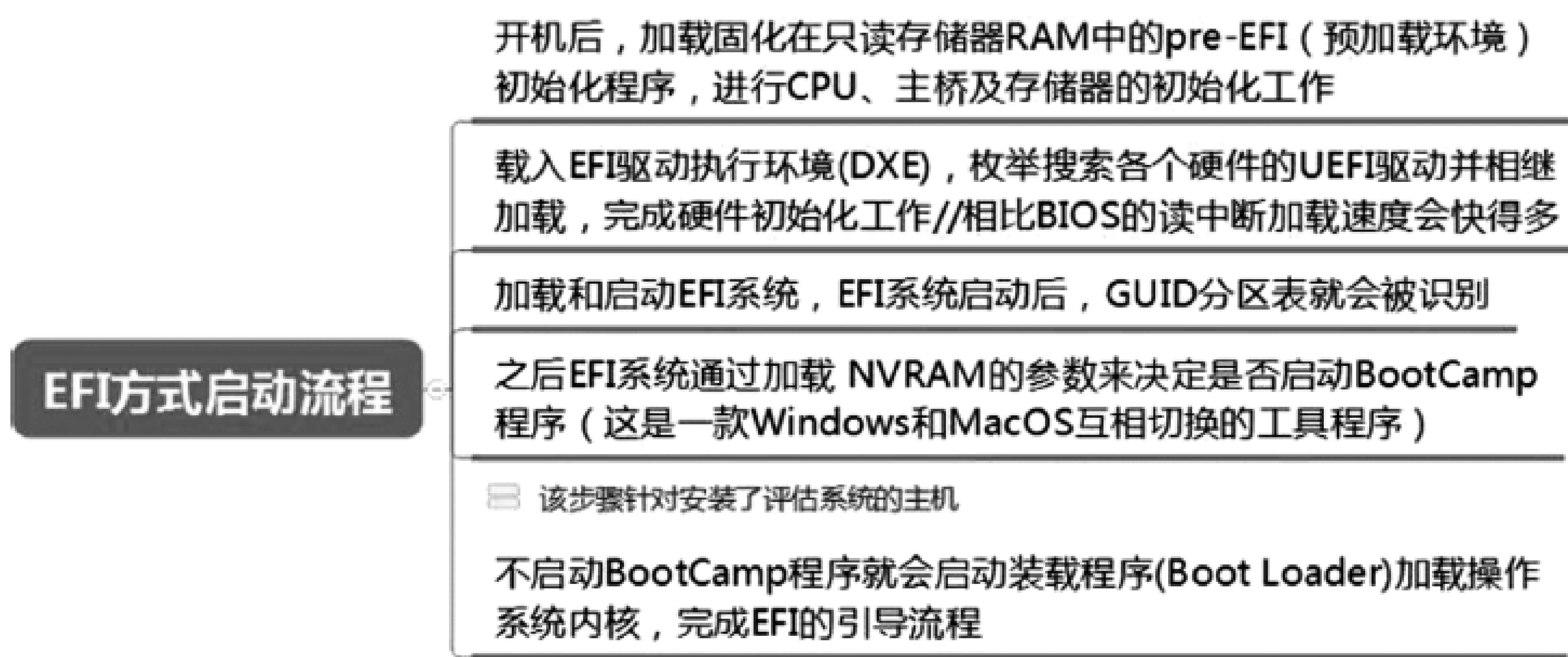


图 13-10　EFI 方式的启动步骤描述

EFI 是一种取代传统 BIOS 方式的技术,不但在引导期提供接口,甚至也为引导之后的系统运行提供了接口。

在 EFI 方式的启动流程中,最后一步是加载操作系统内核,这个过程与 BIOS 方式的“NTLDR 阶段”基本一致,这里不再赘述。

13.2.3　UEFI + GPT 方式

UEFI 方式的系统启动过程是目前计算机系统最先进、最安全的启动过程。

BIOS 方式启动时,操作系统的 Boot Loader 存放在 MBR 中。而 MBR 充其量也就是一个 512 字节的扇区,能有多大? 所以 Boot Loader 代码的容量很受限制。但是 UEFI 引入了一个新的系统分区 ESP(EFI System Partition),这个分区存储 Boot Loader 和 EFI 驱动,容量大大

增加。计算机启动时，UEFI 固件先从 ESP 中加载所需的硬件驱动，再执行 Boot Loader 指定的操作系统。如图 13－11 所示，UEFI 方式的启动流程包含了 7 个阶段。

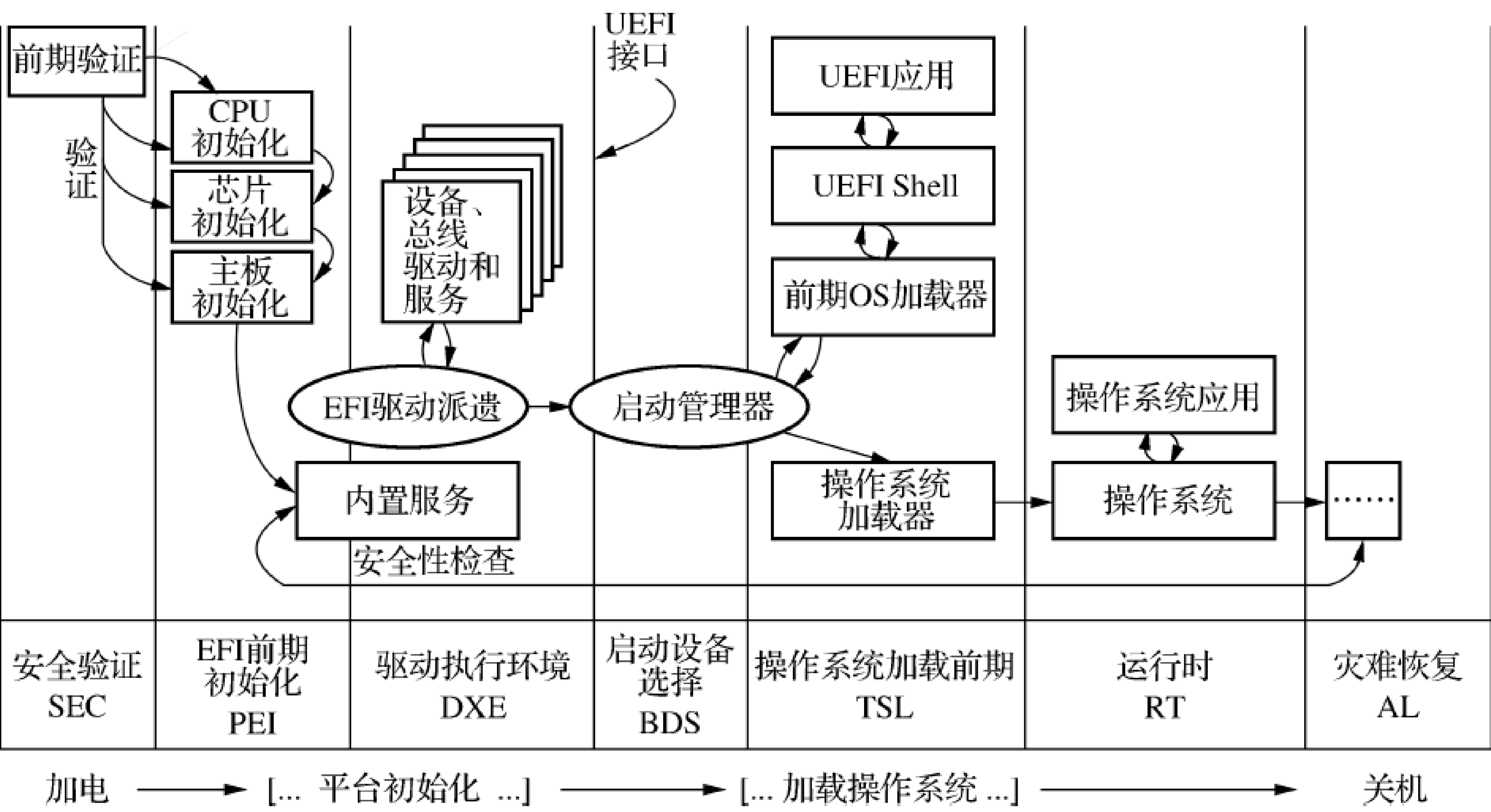


图 13－11　UEFI 方式的系统启动流程和操作系统加载的 7 个阶段

➢ **SEC 阶段**：这个阶段以汇编代码为主，完成的工作主要包括获取 CPU 检测结果，找到 PEI 的二进制代码，并跳转到 PEI 执行入口。

➢ **PEI 阶段**：这个阶段以 C 语言代码为主，完成的工作主要包括 CPU 初始化、内存初始化、芯片和主板初始化，并将上述初始化结果通过 HOB(Hand-off Block，是 PEI 阶段向 DXE 阶段传递系统信息的手段，本质上是一系列连续的内存结构体)传递到下一阶段，如图 13－12 所示。

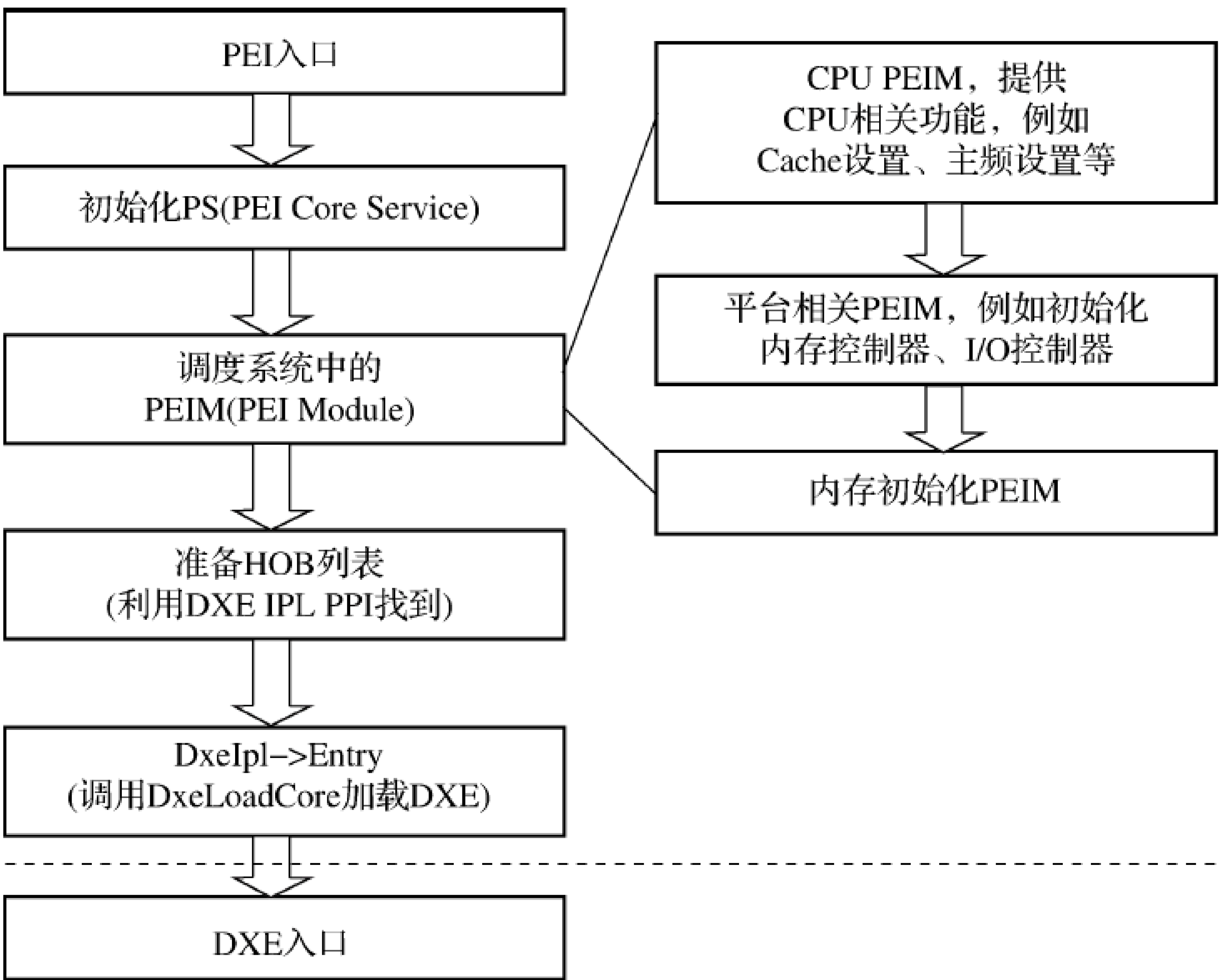


图 13－12　PEI 阶段的执行流程

- **DXE 阶段**:这个阶段运行若干驱动模块,每个模块完成不同的工作,有的是提供功能服务,有的则初始化硬件,如图 13-13 所示。
- **BDS 阶段**:这个阶段生成和枚举启动项,并选择其中一个启动项加载操作系统,同时也负责一些硬件初始化工作。

这 4 个阶段完成,后面就是加载和运行操作系统了,当然这个过程是可以有 UEFI 参与的。

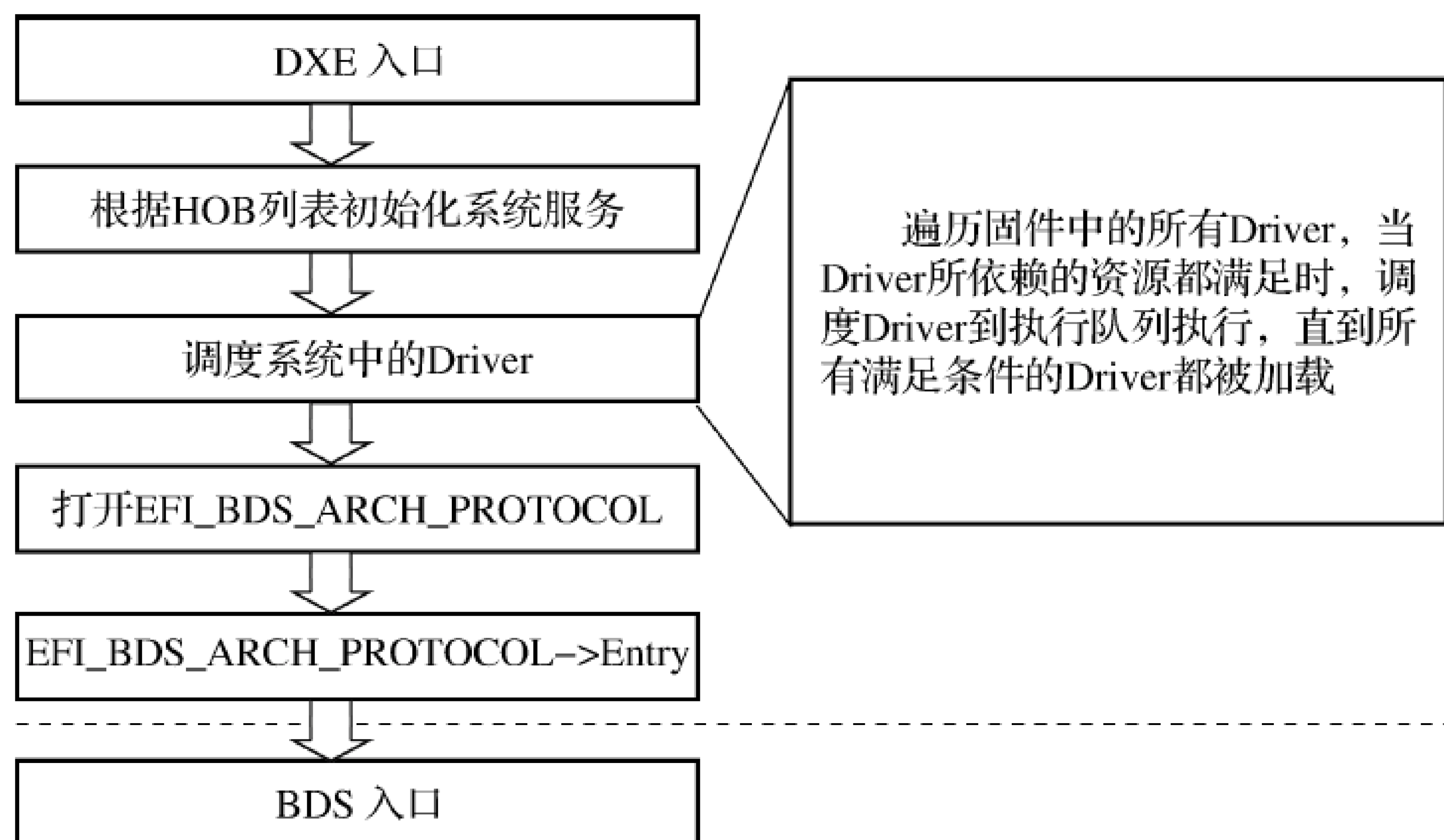


图 13-13　DXE 阶段的执行流程

图 13-14 是对 UEFI 方式的启动步骤的描述,从中可以看出,UEFI 启动方式的一大特点就是增加了安全验证阶段,并且是第一阶段。这个阶段就是瞄准 Rootkit 和 Bootkit 攻击问题而增加的,其设计借鉴了可信计算的思想。UEFI 规范规定了固件可以包含一系列签名,并拒绝运行未签名或签名与固件中包含的签名不一致的 EFI 可执行文件。不过安全启动是个可选项,用户也可以选择关闭这个功能。正因为如此,我们称 UEFI 是目前计算机最安全也最先进的启动方式。

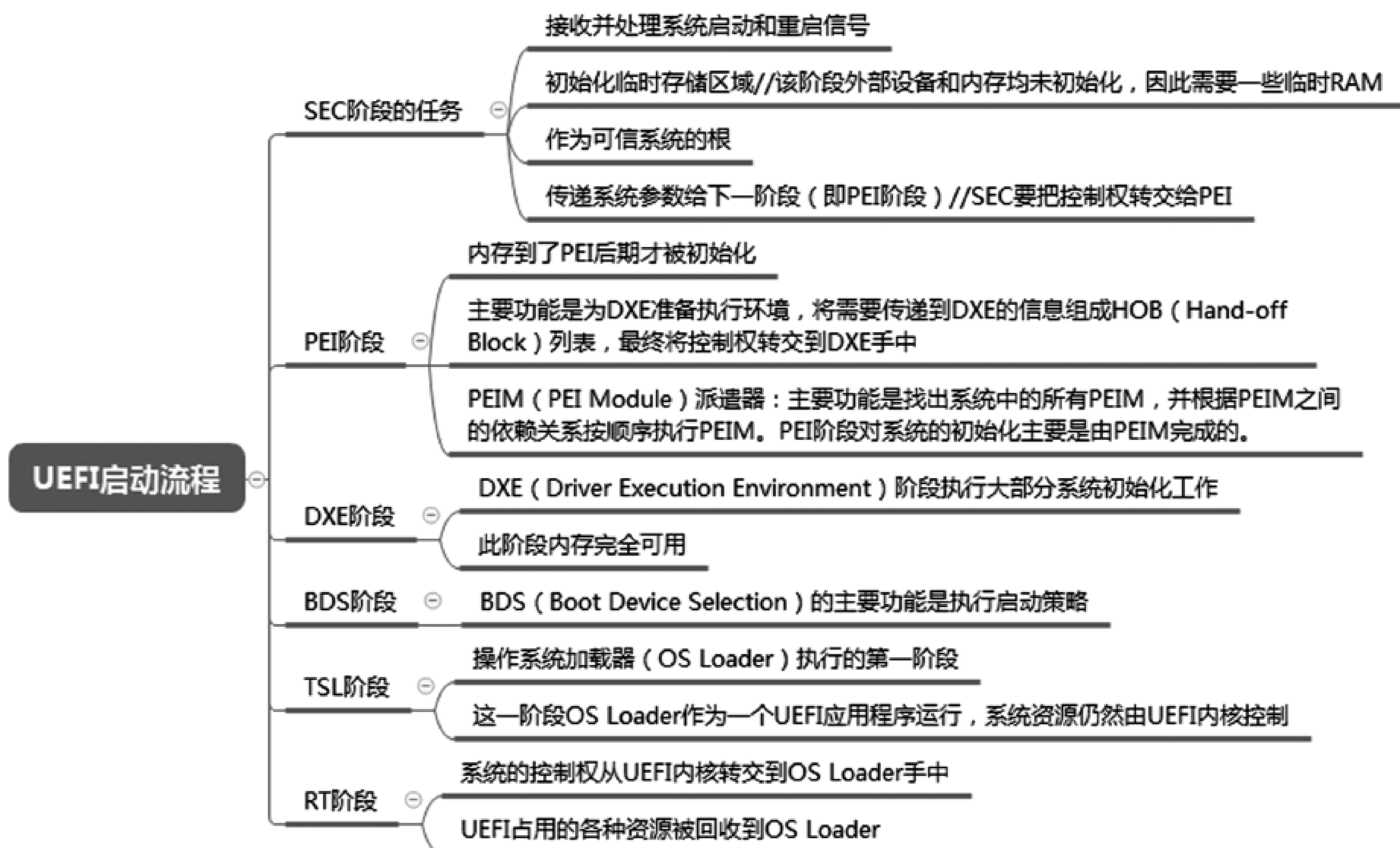


图 13-14　UEFI 方式的启动步骤描述

UEFI 需要额外的存储空间,因此 UEFI BIOS 不再被放在 ROM 中,而是在硬盘中划分出一块 FAT32 格式的扇区(ESP)来存放 UEFI 相关的各类数据,例如 EFI 驱动和应用程序。但如果硬盘发生损坏或 ESP 上的数据被病毒篡改删除,则会引发更大的安全隐患。因此,利用可信计算技术加固 UEFI 驱动和应用程序的安全性是未来的重要课题。

本章小结

本章总结了 Windows 启动过程中各个参与者的分工,并结合三种启动方式描述了 Windows 的启动过程。

第14章　Windows 内存安全机制

安全漏洞是计算机科学自诞生以来就存在的固有顽疾。无论是硬件设计还是软件代码，只要是人写的代码，总会有漏洞存在。应用软件上的一个小 Bug，甚至一个保护机制不是很严密的设计逻辑，轻则使进程运行崩溃，重则成为安全漏洞被外界利用来“做坏事”。而这些坏事多种多样，有的可以监控当前系统的运行，有的可以截取通过键盘输入的内容，更有甚者可以接管操作系统的运行权。

例如 2017 年爆发的“永恒之蓝”（Eternal Blue）系列漏洞，就是利用了 Windows 系统的 SMB 漏洞来获取系统的最高权限，黑客利用这种漏洞开发了 WannaCry 勒索病毒；而 2018 年的“熔断”（Meltdown）和“幽灵”（Spectre）则剑指更底层的 X86/X64 CPU 的分支预测执行机制，利用这种机制进行侧信道攻击以获取缓存中的数据。这种侧信道攻击方法的前两种变体被称为“Meltdown”，第三种变体被称为“Spectre”。

不过，安全漏洞与安全防护机制从来就是此消彼长的“拉锯战”，有漏洞自然就会有弥补漏洞的安全机制存在。随着操作系统的发展和完善，Windows 的安全体系逐渐成熟，形成了若干独立的安全机制。在 Windows 系统安全领域，溢出类漏洞占据了绝大多数，其他则为数据安全（加解密、压缩）、WEB 安全和网络安全等这些细分领域造成的威胁。从本质上讲，操作系统溢出类漏洞生存的土壤是“数据代码和指令代码都不加以区别地存放于内存之中”，大部分软件溢出漏洞都是利用了这种机制；做坏事的指令以看似人畜无害的数据形式存在于内存中，一旦条件成熟就会被触发执行。因此 Windows 防护机制首先面对的就是溢出漏洞。

本章节按照图 14－1 所示的提纲讲述 Windows 系统的内存安全机制，特别是面对溢出类漏洞时 Windows 的种种内存防护机制。在此之前我们先来看一下漏洞与病毒的定义与分类。

14.1　软件漏洞与病毒

14.1.1　软件漏洞

简单地说，软件漏洞是操作系统或应用软件本身存在的缺陷，这种缺陷往往被恶意程序利用而成为威胁寄生的温床。软件漏洞大致分为如下几种。

1. 缓冲区溢出漏洞

缓冲区溢出漏洞通过向进程的缓冲区（例如堆栈）写入超过缓冲区当前栈帧最大数据量的恶意代码使之溢出（例如堆栈溢出，覆盖和超出堆栈的边界上线），打破进程当前栈帧的平衡（修改函数栈帧中的返回地址），从而在函数调用返回时通过返回地址获取进程的执行控制权，如图 14－2 所示。这种漏洞是操作系统和应用软件中存在最为广泛、宿主代码种类最为繁多、占比最大的一种漏洞。通过溢出覆盖了原有堆栈数据的这段代码被称为“跳板”（ShellCode）。

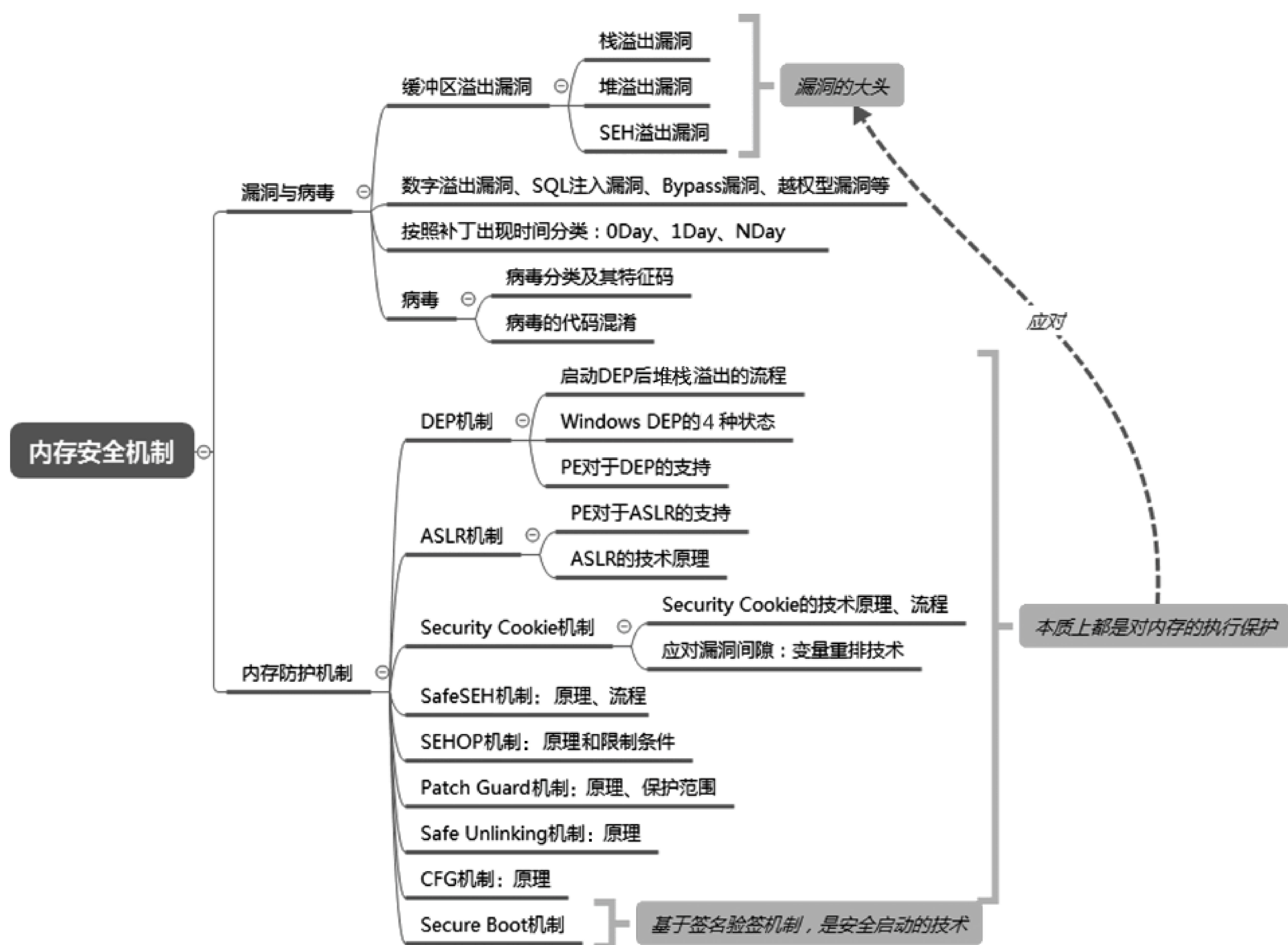


图 14-1 本章提纲

缓冲区溢出漏洞又可分为栈溢出、堆溢出和 SEH（结构化异常处理）溢出等类型，其中 SEH 溢出也是基于栈溢出的原理。由于 SEH 结构是保存在栈中的，因此利用栈溢出淹没 SEH 的异常处理例程指针（Handler 指针），也就是使用 ShellCode 覆盖 Handler，由于发生异常时系统跳转到 Handler 处执行，因此也就跳转到了淹没 Handler 的 ShellCode 某处执行，从而接管了异常发生时的线程执行权。不过从 Windows XP 版本开始系统添加了 SEH 处理的保护机制 SafeSEH，仅通过淹没 Handler 的方式截取进程控制权已经行不通了。表 14-1 列出了堆栈溢出的一般性攻击方法及其保护技术。

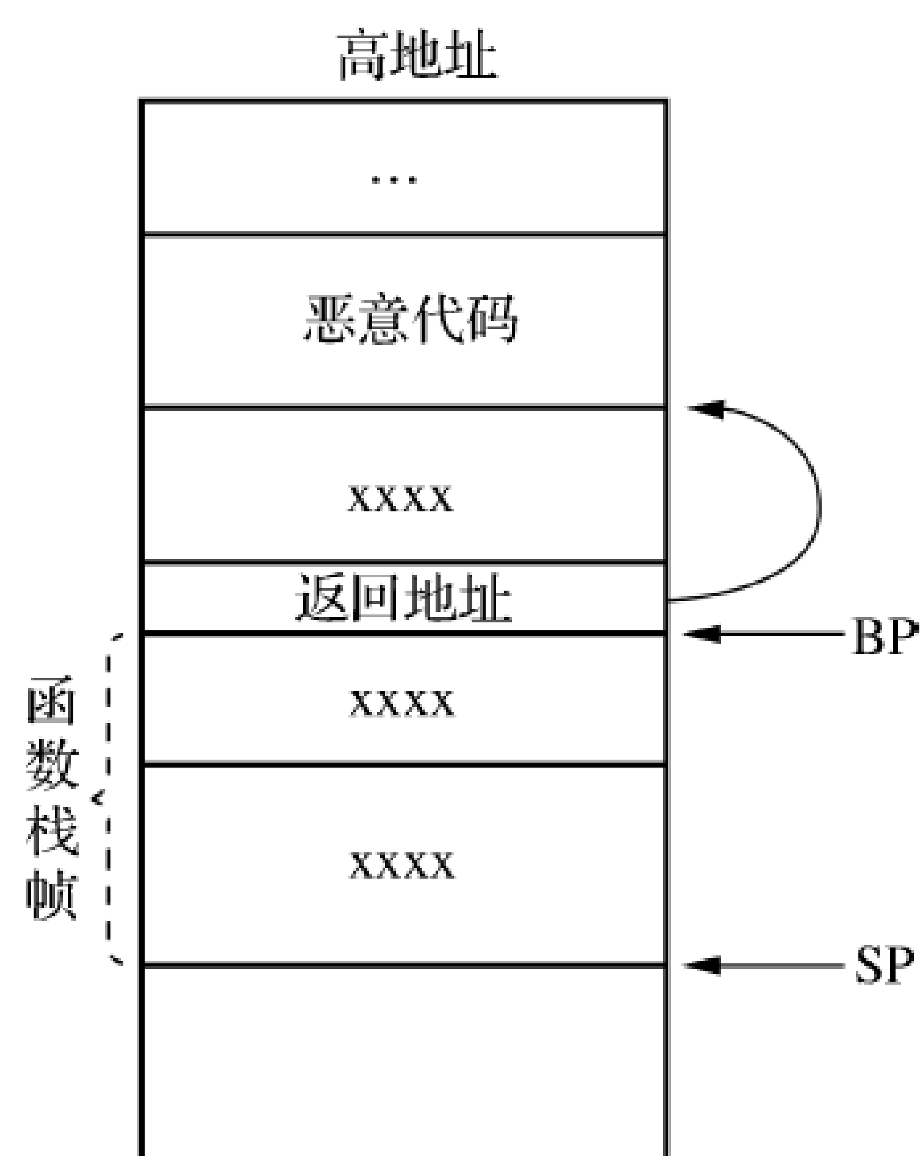


图 14-2 堆栈溢出示意图

表 14-1 堆栈溢出的一般性攻击方法及其保护技术

目的	保护技术
覆盖返回地址	通过 GS 编译选项加以保护，其原理是在函数调用堆栈上插入一个安全 Cookie 以检测返回地址是否被修改
覆盖 SEH 地址	通过 SafeSEH、SEHOP 保护方式加以保护
覆盖本地变量	可被 VC 编译器重新整理和优化，重新调整本地变量在堆栈中的位置

续表 14－1

目的	保护技术
覆盖空闲堆双向链表	通过 safe unlink 方式保护，该方式根据双向链表的特点检查前向和后向指针的有效性
覆盖堆块头	Windows XP 下使用 8 位的 Header Cookie 进行保护，Windows Vista 之后使用 XOR HeaderData
覆盖旁视列表（lookaside list）	Windows Vista 版本之后旁视列表已被移除

缓冲区溢出漏洞还包括格式化字符串漏洞，这是一种针对 print 类函数（例如 printf、sprintf、fprintf 等）的漏洞，因此比较有局限性。由于这类函数在 C 语言中支持可变参数，因此调用者可以自由指定函数参数的数量和类型，而被调用者无法知道在函数调用之前到底有多少参数被压入栈帧当中，从而利用这种机制进行栈溢出造成漏洞，所以格式化字符串漏洞本质上也属于缓冲区溢出漏洞。

2. 数字溢出漏洞

计算机系统中的整数变量有上下界，如果在算术运算中发生越界，就会出现两类整数溢出：

- 超出整数类型的最大表示范围，数字便会由一个极大值变为一个极小值或直接归零，这叫作“上溢”；
- 超出整数类型的最小表示范围，数字便会由一个极小值或者零变成一个极大值，这叫作“下溢”。

在金融行业数字溢出漏洞属于危险等级非常高的漏洞（试想银行卡中的余额“下溢”）。整数溢出不需要改写内存或控制程序执行流程，因此更难以察觉。常见编译器下的整数类型的数值范围如表 14－2 所示。

表 14－2　常见编译器下整数类型的数值范围

编译器类型	数据类型	数据名称	最小值	最大值
VC++ 6.0	unsigned short	无符号短整型	0	65 535
	short	短整型	－32 768	32 767
	unsigned int	无符号整型	0	4 294 967 295
	int	整型	－2 147 483 648	2 147 483 647
	usinged long	无符号长整型	0	4 294 967 295
	long	长整型	－2 147 483 648	2 147 483 647
	unsigned _int64	无符号 64 位整数	0	18 446 774 073 709 500 000
	_int64	64 位整数	－9 223 372 036 854 770 000	9 223 372 036 854 770 000

续表 14－2

编译器类型	数据类型	数据名称	最小值	最大值
VS2012 C#	ushort	无符号短整型	0	65 535
	short	短整型	－32 768	32 767
	uint	无符号整型	0	4 294 967 295
	int	整型	－2 147 483 648	2 147 483 647
	ulong	无符号长整型	0	18 446 744 073 709 500 000
	long	长整型	－9 223 372 036 854 770 000	9 223 372 036 854 770 000

3. SQL 注入漏洞

在没有对用户输入数据的合法性进行判断的情况下，可以提交一段数据库查询代码，根据程序返回的结果，获得某些攻击者想得知的数据，这种漏洞称为 SQL 注入漏洞。SQL 注入漏洞属于 Web 安全领域的漏洞。

4. Bypass 漏洞

也叫作开放重定向(Open Redirect)漏洞，即在访问 B/S 系统时用户输入某个 URL 链接而跳转到攻击者控制的恶意网站。Bypass 漏洞属于 Web 安全领域的漏洞。

5. 越权型漏洞

由于服务器端对客户端提出的数据操作请求权限不加以判定，导致客户端拥有过大的权限来进行增、删、改、查操作，这种漏洞被称为越权型漏洞。越权型漏洞又分为水平越权和垂直越权两种，前者是指攻击者尝试访问与之有相同权限的用户资源，后者则是指攻击者尝试访问更高或更低级别用户的资源。

以上从功能上对漏洞进行了分类，除此之外还可以按照漏洞被发现和破解时间的长短分类，如下所示：

- **0Day 漏洞**：一般是指软件发布后在很短时间内被发现或破解的漏洞，也就是新发现的漏洞，对应的补丁还没有发布。0 即 Zero，0Day 就表示未超过 24 小时，当然这是对短时间的形象表示，并不是说一天内破解的漏洞。
- **1Day 漏洞**：一般是指已经通过各种途径曝光过的漏洞。
- **NDay 漏洞**：一般是指由于计算机用户没有安装相应的漏洞补丁，而被恶意程序利用公开已知的漏洞攻击，这种情况造成的漏洞被称为 NDay 漏洞。

14.1.2　病毒

如果说漏洞是被利用的，那么病毒则是漏洞利用的宿主，具有感染性、潜伏性和表现性三个基本特点。病毒大致按照表 14－3 所示进行分类。

表 14－3　常见病毒分类

病毒分类	病毒特点	主要特性
文件型病毒	通过文件系统感染的病毒统称为文件型病毒	• 感染 PE 文件(. exe 或. com 文件) • 一般位于 PE 文件的头部或尾部 • 执行被感染程序后,病毒一般从入口函数开始执行以获取进程控制权
引导型病毒	通过感染硬盘引导扇区或主引导扇区的病毒统称为引导型病毒	• 先于操作系统加载启动 • 占据主引导扇区或引导扇区的全部或一部分,并将分区表信息或引导记录移到磁盘的其他位置 • 引导时首先获取系统控制权,继而将病毒的主要部分调入内存并驻留在内存高址部分,但最后仍会跳转到主引导记录
宏病毒	利用 Word、Excel 等软件的宏脚本功能进行传播的病毒	• Word 中被嵌入带有恶意行为的宏代码(VBA 代码),当打开带有宏病毒的 Word 文档时宏代码会自动运行 • 利用的是 Office 提供的对于 VBA 脚本的支持 • WPS 不支持宏,不会感染宏病毒
恶意脚本	出于恶意对软件系统进行增加、改变或删除的任何脚本	• 恶意脚本变形简单,混淆机制多样 • 可以动态创建内嵌链接和编码链接内容 • 依赖于浏览器,利用漏洞下载木马 • 包括 Java\Python\JS 等脚本形态
木马程序	在计算机系统中种植的以窃取信息为目的的后门程序	• 具有隐蔽性、欺骗性以保证生存 • 具有自运行、自动恢复的能力 • 一般依赖于 TCP/IP 协议进行通信,会悄悄打开端口
蠕虫病毒	一种利用网络进行复制和传播的病毒	• 利用电子邮件等途径进行传播 • 能够利用漏洞传播自身和自拷贝 • 攻击形式一般是扫描→攻击→复制
其他破坏性程序		

虽然病毒分类五花八门,但是也都具有一定的特征码(包括单一特征码和复合特征码),这些特征码按照宿主介质又可以分为:

➢ **文件特征码**:病毒代码在磁盘上以文件态存在时的特殊指令或数据字符。

➢ **内存特征码**:病毒代码在内存中以运行态存在时的特殊指令或数据字符。

➢ **行为特征码**:病毒在内存中运行时的特殊行为状态,例如内存使用、API 调用、弹/压堆栈等。

病毒为了更好地隐藏自己,一般会采用加壳、花指令或压缩等方式改变自己的特征码形态,当然这些手段也可以用于代码混淆、代码防破解:

➢ **加壳**:通过一系列数学运算将可执行程序或动态链接文件的编码进行改变,以达到缩小程序代码体积或加密程序代码的目的,病毒代码也因此隐藏。由于杀毒软件无法





发现真正的病毒体,利用这种方法可以逃避查杀和隐蔽自身。

➢ **花指令**:一串没有任何实际意义的指令,使程序的分析者或破解者无法清楚正确地反汇编程序的内容,从而达到隐蔽自身真实意图的目的。

➢ **压缩**:病毒有时候会隐藏在压缩包里,这就要求杀毒软件有解压缩探测能力,从而增加了杀毒软件查杀病毒的门槛、步骤和开销,也可以在一定程度上达到隐蔽自身、逃脱查杀的目的。

了解了上述内容后,我们来考察一下 Windows 在系统安全领域的保护机制,特别是基于内存的保护机制。这些机制虽然能够极大地提高漏洞发生和病毒植入的门槛,但仍不能杜绝漏洞和病毒的产生。操作系统中的攻击与保护本来就是此消彼长的,保护方式改进了,攻击方式也必然会推陈出新,从而更加考验保护方式。同时,两者之间的技术手段也不是非黑即白的,这些手段既可以用来保护,亦可以用来攻击。

14.2　DEP 机制

DEP(Data Execution Prevention,数据执行保护)可以在内存页面上做检查以防止存放数据的内存运行代码。DEP 本质上是将数据所在的内存页面标示为"不可执行页",由于当程序溢出时要转入 ShellCode 执行,因此必然是在堆栈缓冲区的内存页面上执行,而此处的页面是数据内存页面,启用 DEP 机制后 CPU 会抛出异常从而禁止在这些数据内存页面执行 ShellCode。

启用和未启用 DEP 机制的堆栈溢出流程比较如图 14－3 所示,开启 DEP 机制后执行 ShellCode 会转到异常处理流程。

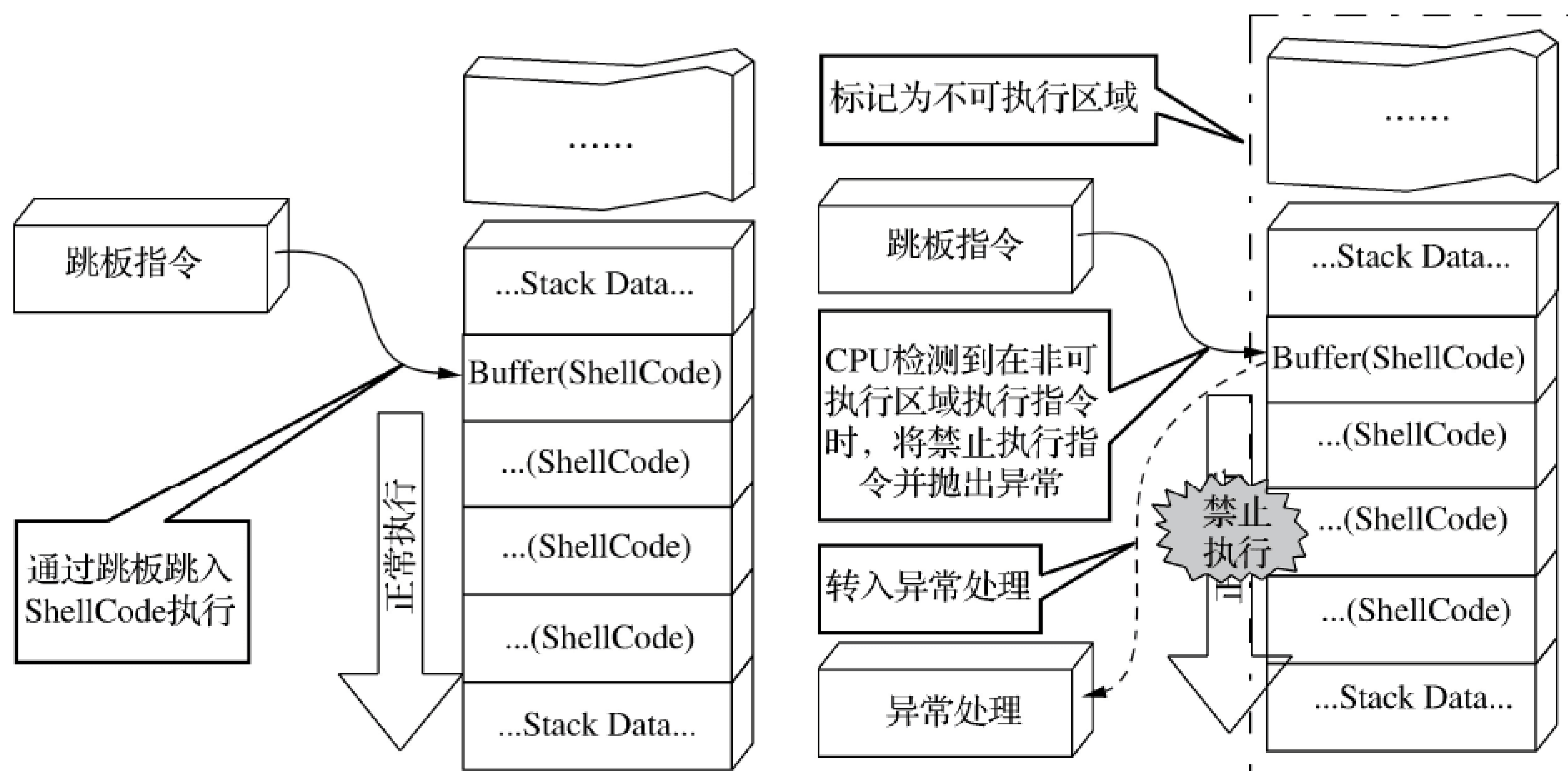


图 14－3　未启用 DEP 机制与启用 DEP 机制后的堆栈溢出流程

数据内存页面主要是指堆栈、内存池等存放数据而非存放指令的内存页面(X86 体系下传递参数时,大多以堆栈为主,函数内部创建的局部变量也是存放在堆栈中,因此堆栈构成

了数据内存页面的主体)。DEP 机制一般是采用硬件机制实现的,与软件实现机制相比其检查效率更高。Intel 和 AMD 的 CPU 都为 DEP 机制做了专门的设计,工作原理也都差不多,其中 Intel 的 DEP 机制称为"Execute Disable Bit(XD)",AMD 的 DEP 机制称为"No-Execute Page-Protection(NX)",简单来说就是由操作系统来设置内存页的 ND/NX 属性,指出哪些内存页面不能被执行,因此在操作系统的内存页面表中要加入一个 ND/NX 标志位,用于标注页面的不可执行属性。

Windows 从 XP2 版本开始支持了 DEP 机制,当然该机制是可以选择性开启的,用户可以自主选择不需要被 DEP 机制保护的服务或程序,除此之外的程序都会被 DEP 机制保护,如图 14-4 所示。

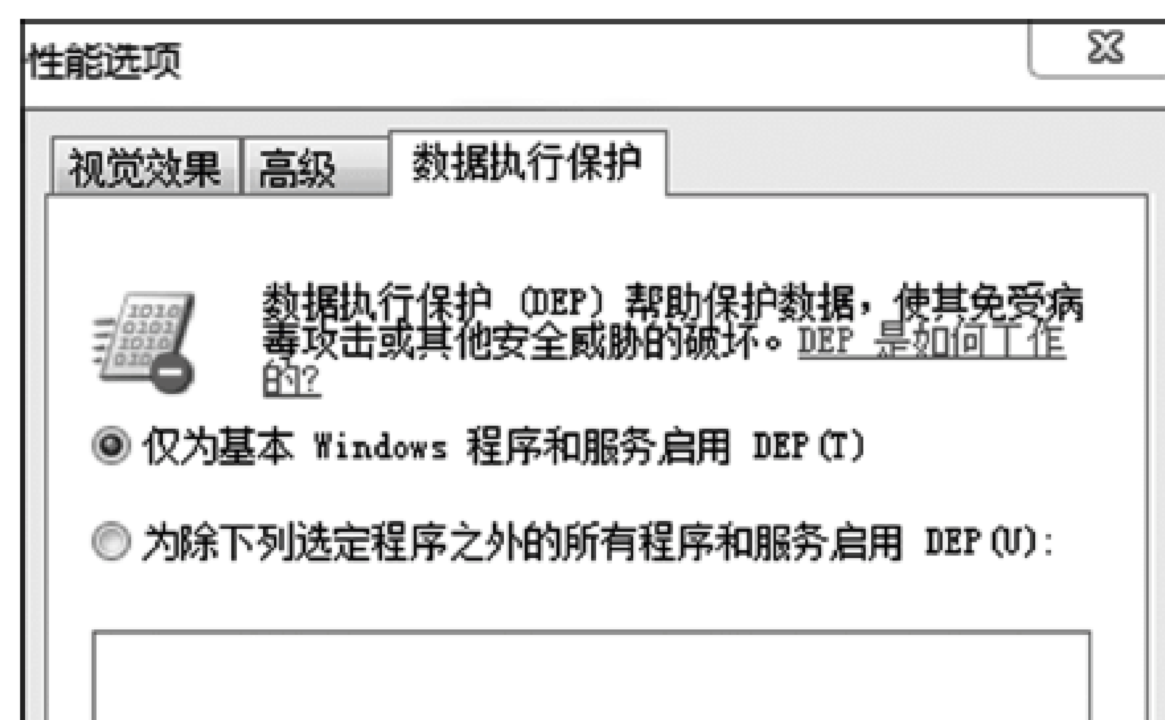


图 14-4 Windows 7 64 位系统下对于 DEP 机制支持的选择

DEP 机制的工作状态分为以下 4 种:

- **Optin 状态**:DEP 仅应用于 Windows 系统组件和服务,Optin 状态一般用于 Windows 桌面系统,同时也是系统的默认状态。
- **Optout 状态**:对图 14-4 中"为除下列选定程序之外的所有程序和服务启用 DEP"列表之外的所有程序启用 DEP,Optout 状态一般用于 Windows 服务器系统。
- **AlwaysOn 状态**:对系统中的所有程序启用 DEP,且 DEP 不可以被关闭,AlwaysOn 状态只支持 64 位系统。
- **AlwaysOff 状态**:对系统中的所有程序禁用 DEP,且 DEP 不可以被开启。

同时,大多数现代编译器也支持对所编译程序代码的 DEP 使能选项,如图 14-5 所示。DEP 使能编译后的 PE 文件可选头结构 IMAGE_OPTIONAL_HEADER 会被 IMAGE_DLLCHARACTERISTICS_NX_COMPAT 标志置位(DllCharacteristics 的 Image is NX compatible 选项被选中),如图 14-6 所示,不过只在 Windows 7 及以上版本中才有效。

映像名称	用户名	CPU	工作设...	峰值工作设...	内存(专用工...	提交大小	数据执行保护
Foxmail.exe *32	tanzhe	00	26,916 K	27,036 K	5,740 K	18,780 K	启用
Foxmail.exe *32	tanzhe	00	30,744 K	122,704 K	21,300 K	66,896 K	启用
Foxmail.exe *32	tanzhe	00	53,280 K	97,140 K	27,500 K	41,528 K	启用
HuaweiHiSuiteServ...	SYSTEM	00	9,440 K	9,676 K	2,756 K	3,464 K	启用
ibmpmsvc.exe	SYSTEM	00	4,728 K	4,728 K	1,448 K	1,748 K	启用
igfxCUIService.exe	SYSTEM	00	7,072 K	7,192 K	1,780 K	1,992 K	启用
igfxEM.exe	tanzhe	00	66,920 K	87,308 K	11,048 K	14,068 K	启用
JisuPdf.exe *32	tanzhe	00	340,924 K	357,532 K	324,772 K	326,780 K	启用
lenovodrvsrv.exe *32	SYSTEM	00	20,048 K	20,136 K	10,848 K	18,916 K	停用
Locator.exe	NETWO...	00	2,552 K	2,568 K	880 K	952 K	启用
lsass.exe	SYSTEM	00	13,580 K	13,756 K	4,896 K	5,488 K	启用
lsm.exe	SYSTEM	00	4,928 K	4,956 K	1,956 K	2,912 K	启用
lvvsst.exe	SYSTEM	00	6,688 K	6,760 K	2,336 K	2,528 K	启用
Maxthon.exe *32	tanzhe	00	102,376 K	119,900 K	77,096 K	85,848 K	停用
Maxthon.exe *32	tanzhe	00	39,092 K	46,976 K	22,196 K	25,004 K	停用
Maxthon.exe *32	tanzhe	00	70,780 K	78,988 K	37,896 K	42,716 K	停用
Maxthon.exe *32	tanzhe	00	212,504 K	311,304 K	140,692 K	198,900 K	停用
Maxthon.exe *32	tanzhe	00	222,580 K	487,324 K	180,900 K	230,712 K	停用
Maxthon.exe *32	tanzhe	00	350,352 K	425,704 K	316,624 K	369,100 K	停用
Maxthon.exe *32	tanzhe	00	26,716 K	28,488 K	11,040 K	14,504 K	停用
mDNSResponder.exe	SYSTEM	00	6,800 K	6,836 K	2,740 K	2,924 K	启用

图 14-5 Windows 7 64 位系统下启用和未启用 DEP 机制的进程列表

图 14－6　PE 可选头结构对于 DEP 机制的支持（DllCharacteristics）

不过，DEP 机制也不是万能的。首先，版本较老的 CPU 和操作系统不支持 DEP 机制；其次，Windows 也不是对所有进程都采用 DEP 机制进行保护，因为采用 DEP 机制后许多依赖于内存页面执行的非恶意进程就无法正常工作了；另外，进程中一旦有一个模块不支持 DEP 机制，则整个进程就不能使能 DEP。因此 DEP 机制是一柄双刃剑，不具有“放之四海而皆准”的普适性。最后，操作系统也预留了使能 DEP 的接口（可使用 NtSetInformationProcess 函数中的参数 ExecuteFlags 指定），这就为恶意进程关闭 DEP 提供了突破机会。例如在 KPROCESS 结构中，当 DEP 被启用时，KPROCESS. ExecuteDisable 被置位；当 DEP 被禁用时，KPROCESS. ExecuteEnable 被置位。

另一方面，为了应对 DEP 机制给覆盖返回地址带来的不利影响，ret2libc、ret2plt 和 ROP 攻击等多种多样的攻击技术也被提了出来。其中 ROP 就是面向返回语句的编程方法（Return-Oriented Programming），它借用 libc 代码段里面的 ret/call/jmp/等指令段一个接一个地跳转以执行某项功能，从而避开 DEP 等保护措施。

14.3　ASLR 机制

ASLR（Address Space Layout Randomization，地址空间布局随机化）机制允许程序加载的时候不再按照 PE 文件头中建议的固定基址进行映射（如图 14－7 所示），而改为按照随机地址加载和映射。从 Windows Vista 版本后 ASLR 机制真正成熟起来。ASLR 机制需要操作系统和进程模块自身的双重支持。

PE 文件可选头结构的 DllCharacteristics 属性可以决定这个 EXE 文件是否使用 DEP/ASLR 技术，定义如下：

➢ IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE：0x0040，ASLR 机制使能。

➢ IMAGE_DLLCHARACTERISTICS_NX_COMPAT：0x0100，DEP 机制使能。

如图 14－8 所示，Windows 7 系统下的 kernel32 模块中 DllCharacteristics 的可选属性中，IMAGE_DLLCHARACTERISTICS_NX_COMPAT 的值为 0x0100，表示支持 DEP 机制；IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 的值为 0x0040，表示支持 ASLR 机制，两者的组合键是 0x0140。因此 kernel32 模块对 DEP 和 ASLR 机制都是支持的。

kernel32.dll

Member	Offset	Size	Value	Meaning
Magic	000000F8	Word	020B	PE64
MajorLinkerVersion	000000FA	Byte	09	
MinorLinkerVersion	000000FB	Byte	00	
SizeOfCode	000000FC	Dword	0009AE00	
SizeOfInitializedData	00000100	Dword	00081000	
SizeOfUninitializedData	00000104	Dword	00000000	
AddressOfEntryPoint	00000108	Dword	00015090	.text
BaseOfCode	0000010C	Dword	00001000	
ImageBase	00000110	Qword	0000000078D20000	

图 14-7　PE 文件头中建议的加载基址（ImageBase）

23	DllCharacteristics	0000013E	2	0140
24	SizeOfStackReserve	00000140	8	0000000000004
25	SizeOfStackCommit	00000148	8	0000000000000
26	SizeOfHeapReserve	00000150	8	0000000000010
27	SizeOfHeapCommit	00000158	8	0000000000000
28	LoaderFlags	00000160	4	00000000
29	NumberOfRvaAndSizes	00000164	4	00000010

图 14-8　kernel32.dll 的 DllCharacteristics 属性中对于 DEP/ASLR 机制的标志

ASLR 机制分为以下几个部分：

- **映像随机化**：PE 文件映射到内存中时其加载的虚拟基址是随机的，每次系统重启后加载 PE 文件都是不一样的地址（注意，是系统重启而不是进程重启），这个随机加载地址是在操作系统启动时确定的，Windows 支持以配置的方式选择是否开启映像随机化。不过映像随机化做得不是很彻底，以 32 位虚拟地址为例，其改变的只是映像基址的高 16 位，低 16 位是不变的，这就为枚举式猜测映像加载基址提供了可能。
- **堆栈随机化**：进程每次启动时堆栈的基址都是随机的，如图 14-9 所示。堆栈的地址是在进程启动而非系统启动的时候确定的，因此每次重启进程时堆栈的地址都不一样。堆栈地址不一样，堆栈中的临时变量、参数和返回地址等内容的位置也就不一样了，这就为跳板植入带来了门槛。
- **进程/线程环境块随机化**：PEB 和首个 TEB 的基址不再被固定为 0x7FFDF000 和 0x7FFDE000，进程/线程每次启动时 PEB 和 TEB 也具有随机性，这种随机化从 Windows XP 的 SP2 版本起就已经支持了。

为了破解 ASLR 机制，攻击技术也发生了很大的改变，堆喷射（Heap Spray）就是专门针对 ASLR 的一种攻击技术，浏览器的溢出攻击正是利用了堆喷射。但 Heap Spray 只是一种辅助技术，需要结合其他的栈溢出或堆溢出等技术才能发挥作用。其原理是在 ShellCode 的

```
寄存器 (FPU)
EAX 74B5343B kernel32.BaseThreadInitThunk
ECX 00000000
EDX 00411DDF ZXConfig.<ModuleEntryPoint>
EBX 7EFDE000
ESP 0018FF8C ASCII "M4µt"
EBP 0018FF94
ESI 00000000
EDI 00000000

EIP 00411DDF ZXConfig.<ModuleEntryPoint>
```

```
0018FFC4  FFFFFFFF  SEH 链的末端
0018FFC8  773C4DCD  SE 处理程序
0018FFCC  0051358E
0018FFD0  00000000
0018FFD4  0018FFEC
0018FFD8  773897D5  从返回 ntdll.773897DB 自 ntdll.773897D5
0018FFDC  00411DDF  ZXConfig.<ModuleEntryPoint>
0018FFE0  7EFDE000
0018FFE4  00000000
0018FFE8  00000000
0018FFEC  00000000
0018FFF0  00000000
0018FFF4  00411DDF  ZXConfig.<ModuleEntryPoint>
0018FFF8  7EFDE000
0018FFFC  00000000  堆栈基址及其内容
```

图 14－9　某用户态进程的堆栈指针和堆栈基址

前面加上大量的 slide code(滑板指令),组成一个注入代码段。然后向系统申请大量内存,并且反复用注入代码段来填充。这样就使得进程的地址空间被大量的注入代码所占据,再结合其他的漏洞攻击技术控制程序流,使得程序执行到堆上,最终导致 ShellCode 的执行。

14.4　Security Cookie 机制

Security Cookie(也称为 GS 机制)是 Windows 专门针对栈溢出提出的一种安全机制,该机制需要得到编译器的支持才能正常使用。微软从 VS(Visual Studio) 2003 版本开始支持该机制(GS 校验选项),启用了 GS 校验选项的函数前后会添加额外的处理代码,用于生成伪随机数的 Cookie 并放入当前 PE 模块的.data 区段,当本地变量初始化时首先会向栈中插入这个 Cookie,如图 14－10 所示。因此,Security Cookie 是通过向进程的.data 区段写入 Cookie 的办法来存储原有栈帧中二进制代码的可信性的,在函数返回的时候通过验证这个可信性来判断是否遭受了溢出攻击。

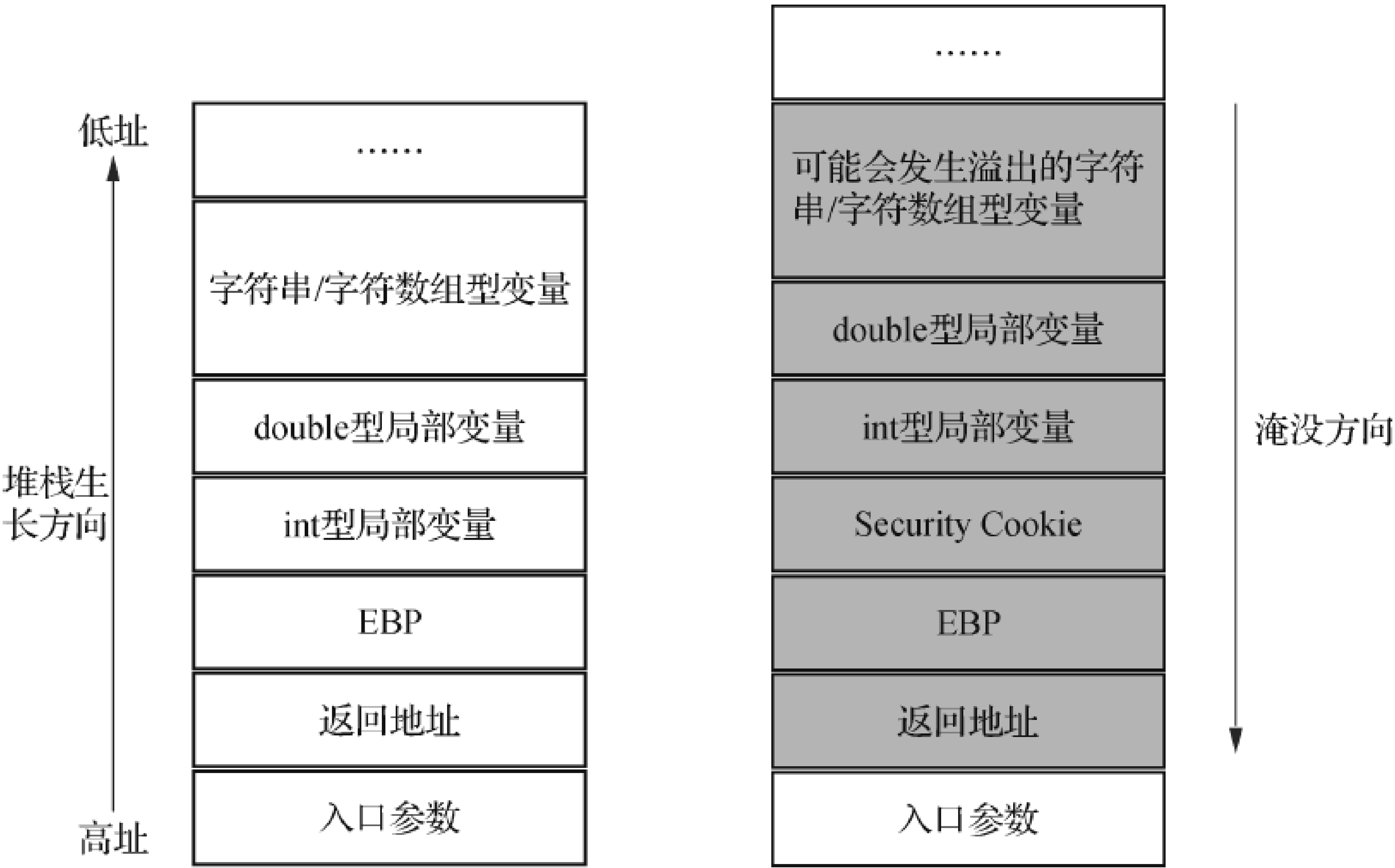


图 14－10　Security Cookie 机制逻辑视图

采用 Security Cookie 机制验证的具体做法如下所示：

（1）程序启动时，首先读取.data 区段的第一个 DWORD 作为种子，然后与各种元素（例如时间戳、进程 ID、线程 ID 等）进行异或（XOR）加密。

（2）将上述加密后的种子再次写入.data 区段的第一个 DWORD 处，替换原来的第一个 DWORD 值。

（3）函数调用执行前将上述加密后的种子取出，并与当前 ESP 寄存器中的值进行异或计算，得到的结果压入栈中 EBP 的前面（栈生长方向的低址部分），我们称这个值为 Security Cookie。

（4）当函数执行返回时，若在当前的栈帧中遭遇栈溢出，则必然将栈帧中的返回地址淹没（否则无法通过返回地址接管线程控制权），也必然淹没到步骤（3）中被压入 EBP 前面的 Security Cookie。因此返回之前把 Security Cookie 取出并与 ESP 寄存器中的值进行异或计算后再调用 security_check_cookie 函数进行检查（与.data 区段里的种子进行比较）。如果两者一致，证明没有被栈溢出覆盖，则返回原函数继续执行；如果校验失败，证明已被覆盖修改，此时应将程序终止。整个过程参见图 14－11。

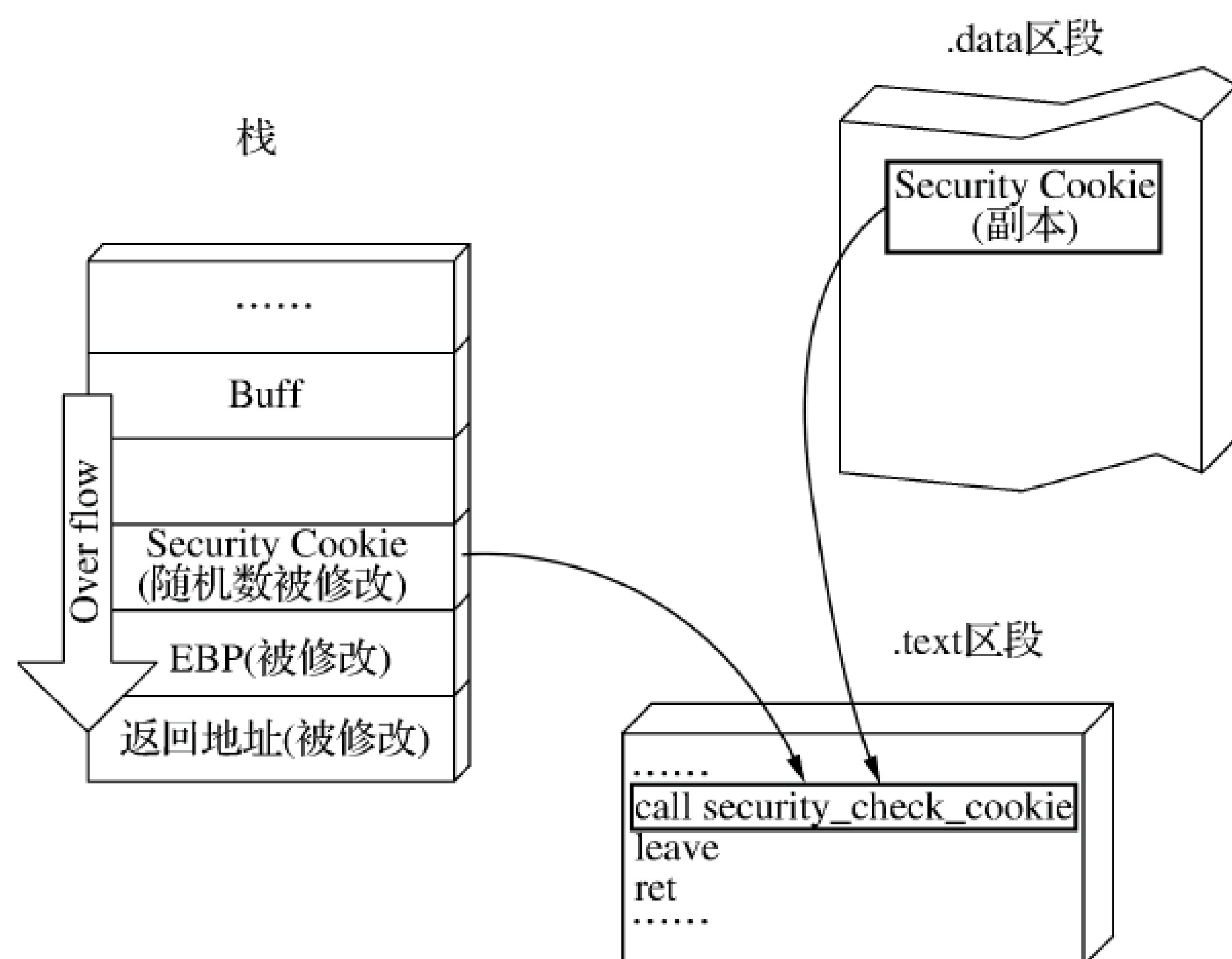


图 14－11　Security Cookie 机制的验证过程

前文说过，栈溢出机制奏效的前提是栈帧淹没到返回地址，也就是将栈帧中的返回地址替换为其他恶意模块的入口地址从而实现跳转。但在一些特殊情况下淹没的深度比较浅，不会淹没到返回地址处，甚至不会淹没到栈帧中的 Security Cookie 处（例如用于溢出的字符串的高址方向还有其他变量），不淹没 Security Cookie 的情况下安全校验机制不会奏效，从而造成了漏洞间隙。

为了应对这种漏洞间隙，从 VS 2005 版本开始，编译器推出了变量重排技术，即编译时根据局部变量的类型对变量在栈帧中的位置进行调整，将字符串变量移动到栈帧的高址区域（紧邻 Security Cookie 的位置），如图 14－12 所示，在发生栈溢出时会增大 Security Cookie 被覆盖的几率。当然，如果用于溢出的字符串的高址方向还有其他的字符串存在，则仍然有

可能由于淹没深度较浅而没有淹没到 Security Cookie，如图 14－12 中最右侧。

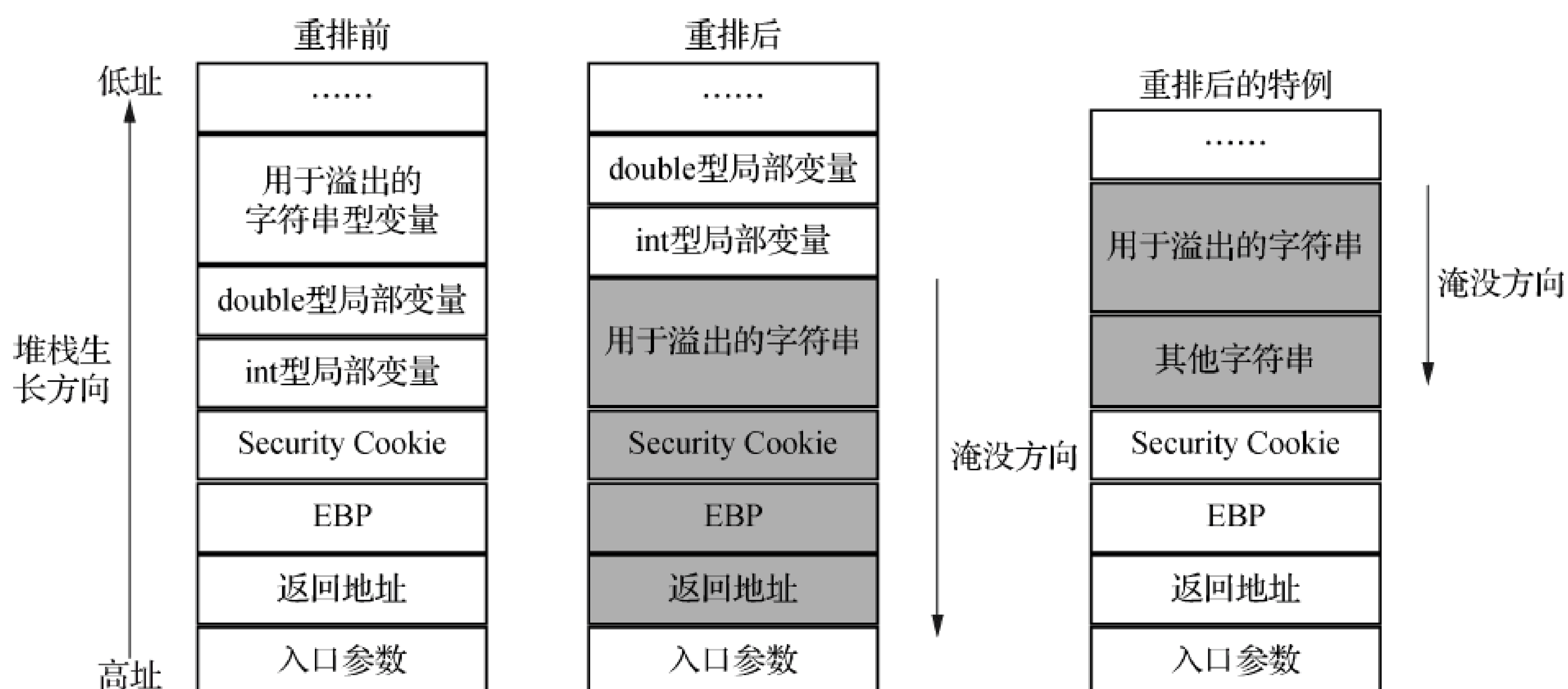


图 14－12　变量重排及潜在漏洞间隙示意图

14.5　SafeSEH 机制

通过栈溢出的方式淹没 SEH 处理函数是堆栈溢出漏洞攻击的经典方式，因此对于 SEH 处理函数地址的保护和校验就显得尤为重要。SafeSEH 就是一种对 SEH 处理函数进行校验的机制，其原理是：

➢ 编译器在编译程序时将所有异常处理函数的入口地址加密写入安全 SEH 表，该表存放在进程的某个区段中。

➢ 线程在调用异常处理例程时对要调用的异常处理函数与安全 SEH 表中的函数指针进行比较和匹配，如果未匹配成功则证明 SEH 处理函数已被覆盖。

SafeSEH 机制的工作流程如图 14－13 所示。

SafeSEH 机制需要操作系统和编译器共同支持才能真正起作用，VS 2003 及更高版本的编译器默认启用该机制（通过/SafeSEH 链接选项）。当进行异常处理时，首先从异常处理的公共分发函数 RtlDispatchException 开始执行，启用了 SafeSEH 机制的分发函数进行以下检查判断：

➢ 若异常处理链不在当前线程的栈中，则终止异常处理（SEH 链都是存放在栈中的）。

➢ 若异常处理例程的指针指向当前栈中某个地址，证明该地址为 ShellCode，则终止异常处理。

上述两项检查完成后再调用 RtlIsValidHandler 方法进行异常处理的有效性检查，包括处理例程是否存在于当前加载模块的内容地址空间中、处理例程是否存在于不可执行的内存页中等，最终 RtlIsValidHandler 只允许在以下几种情况下执行异常处理例程：

➢ 异常处理函数指针位于加载模块内存范围外，且 DEP 未启用。

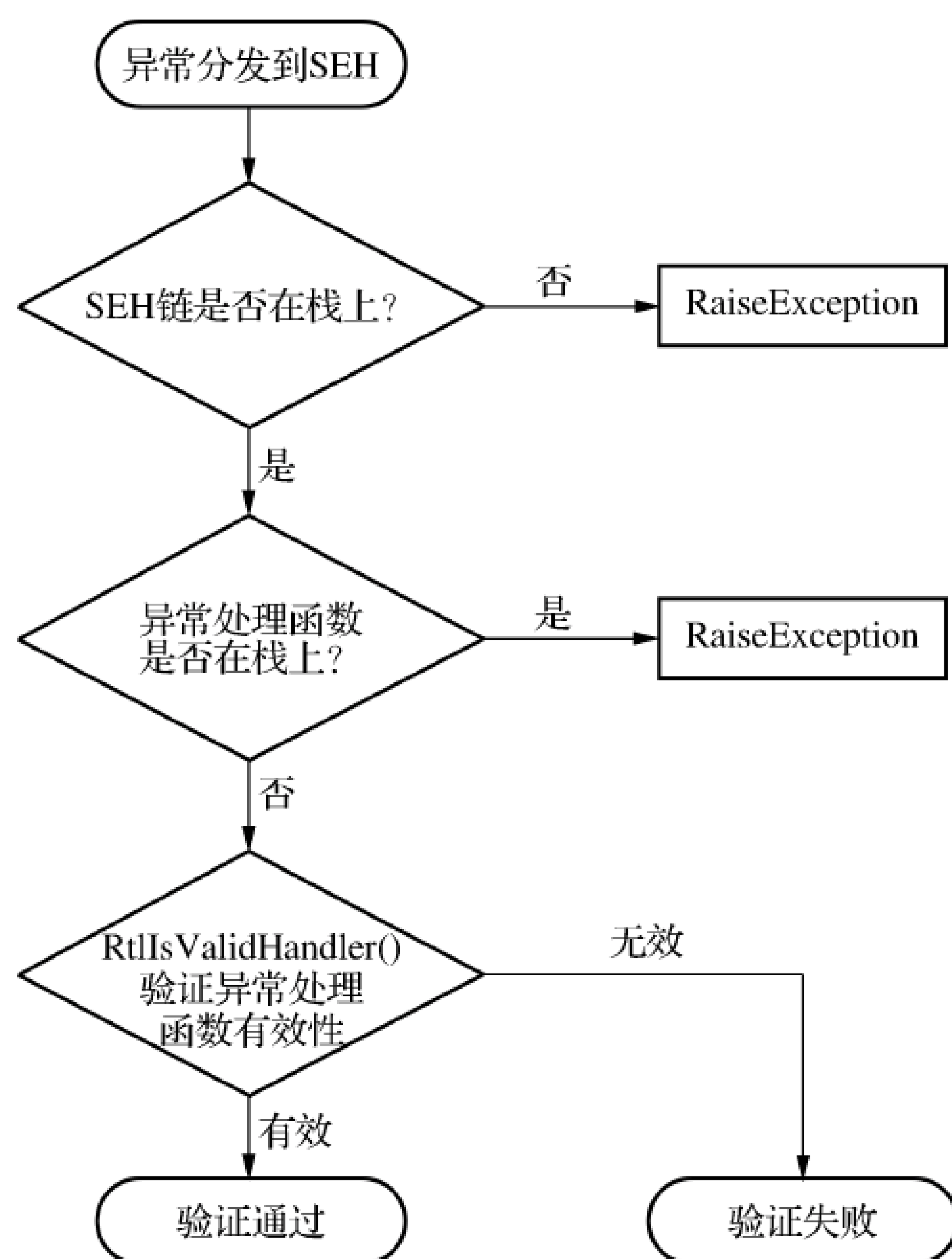


图 14－13　SafeSEH 机制的工作流程

➢ 异常处理函数指针位于加载模块内存范围内，且相应模块未启用 SafeSEH，其状态也不是纯 IL(包含 IL 标志的.NET 中间语言程序，IL 即中间语言，Intermediate Language)状态。

➢ 异常处理函数指针位于加载模块内存范围内，且相应模块启用了 SafeSEH 机制，函数地址也在安全 SEH 表中，如图 14－14 所示。

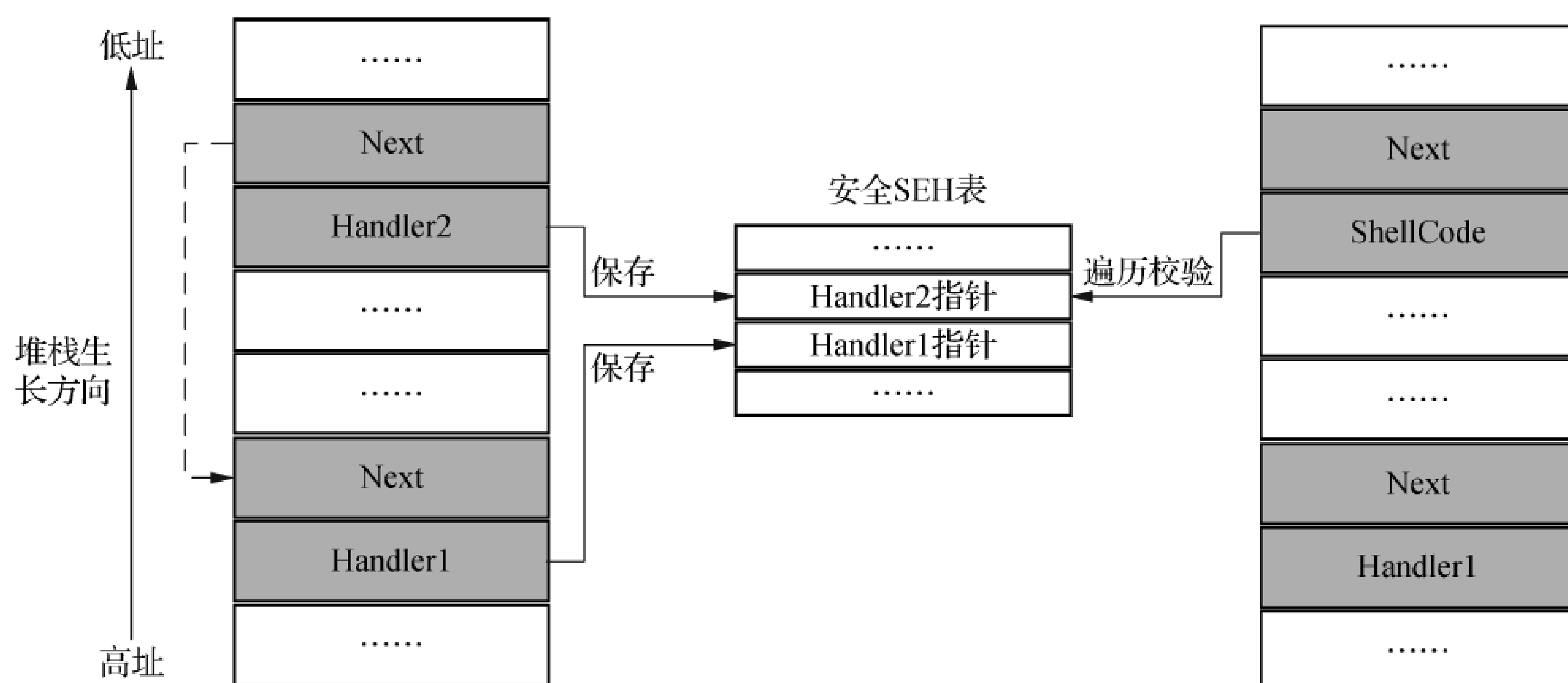


图 14－14　SafeSEH 机制的基本原理

14.6　SEHOP 机制

SEHOP(Structured Exception Handler Overwrite Protection,结构化异常处理覆盖保护)是微软在 Windows Vista 及以上版本中采取的一种较为严苛的 SEH 保护机制,用于保护 SEH 不被栈溢出覆盖。其基本原理是检查 SEH 链的完整性,即检查链中最后一个异常处理(通常是系统默认的异常处理函数)是否为系统默认“兜底”的那个异常处理函数,如图 14－15 所示。这个原理基于下列依据:

➢ 被淹没的 SEH 结构,其 Next 指针也会被覆盖,从而使当前节点失去向下寻找的线索。

➢ SEH 链中最后一个异常处理节点的 Handler 指向的是系统默认的异常处理函数(即 ntdll. dll 中的 UnhandledExcetionFilter),该函数的地址是固定的。

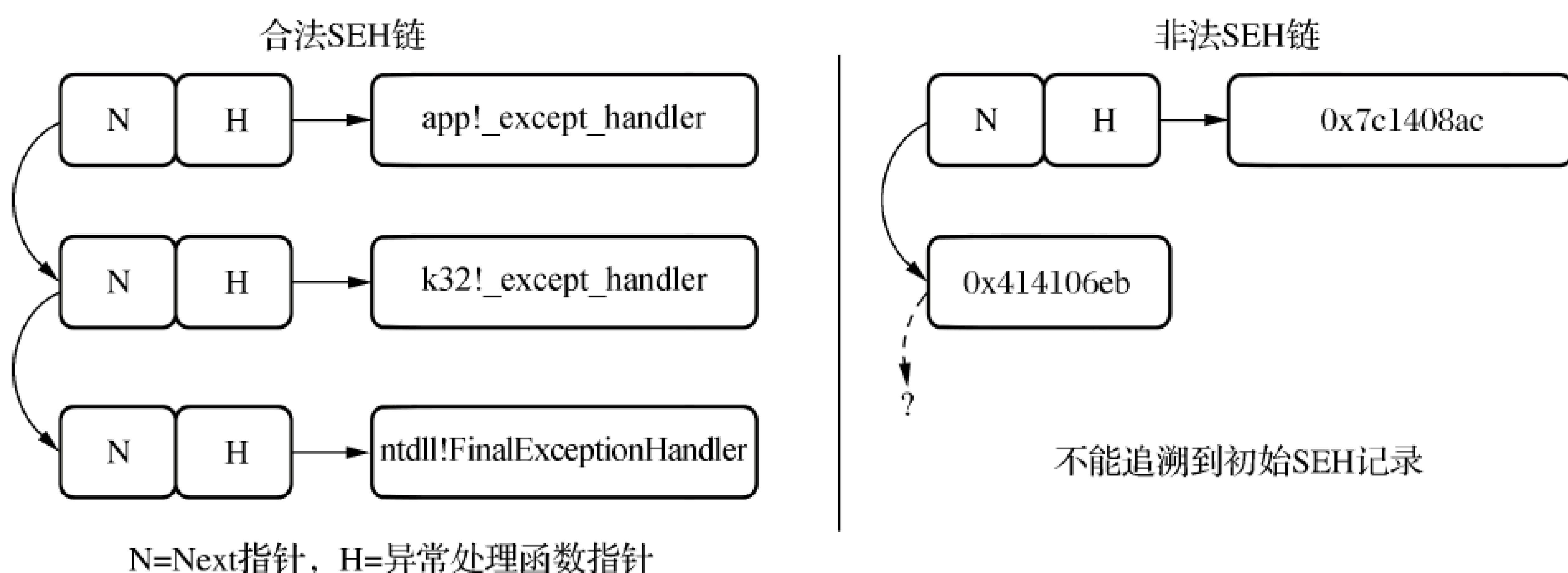


图 14－15　SEHOP 机制的基本原理

SEHOP 机制的实现也是在异常处理公共函数 RtlDispatchException 中。不过 SEHOP 机制要求对于 SEH 做出一些限制条件,包括:

➢ SEH 结构都必须在栈上。

➢ 所有的 SEH 结构都必须是 4 字节对齐的(32 位下)。

➢ SEH 结构中的 Handler(处理函数地址)必须不在栈上。

➢ 最后一个 SEH 结构的 Handler 必须是 ntdll! FinalExceptionHandler 这个函数。

➢ 最后一个 SEH 结构的 Next SEH 指针必须为特定值 0xFFFFFFFF(最后一个节点的默认值)。

14.7　Patch Guard 机制

Patch Guard(内核补丁保护程序,简称 KPP)是 Windows Vista 及以上版本采用的一种内核保护机制,应用于 64 位系统。Patch Guard 能够有效防止内核模式驱动对于 Windows 内核内容的改动或替换,采用 Patch Guard 机制后第三方软件将无法再给 Windows Vista 及以上版本系统的内核添加任何补丁(例如 SSDT HOOK、IDT HOOK 等均无法实施)。

Patch Guard 机制的工作原理是定期检查内存中受保护的内核系统结构是否被修改，一旦发现不一致的地方就以 BSOD（死亡蓝屏）的方式进行阻断，是 Windows 系统中防御 Rootkit 的终极手段，其初始化及工作过程如图 14－16 所示。

Patch Guard 保护的内核组件包括：

➢ 系统模块，包括 NTOS、NDIS 和 HAL；

➢ 系统服务描述符表（SSDT）；

➢ 全局描述符表（GDT）；

➢ 中断描述符表（IDT）。

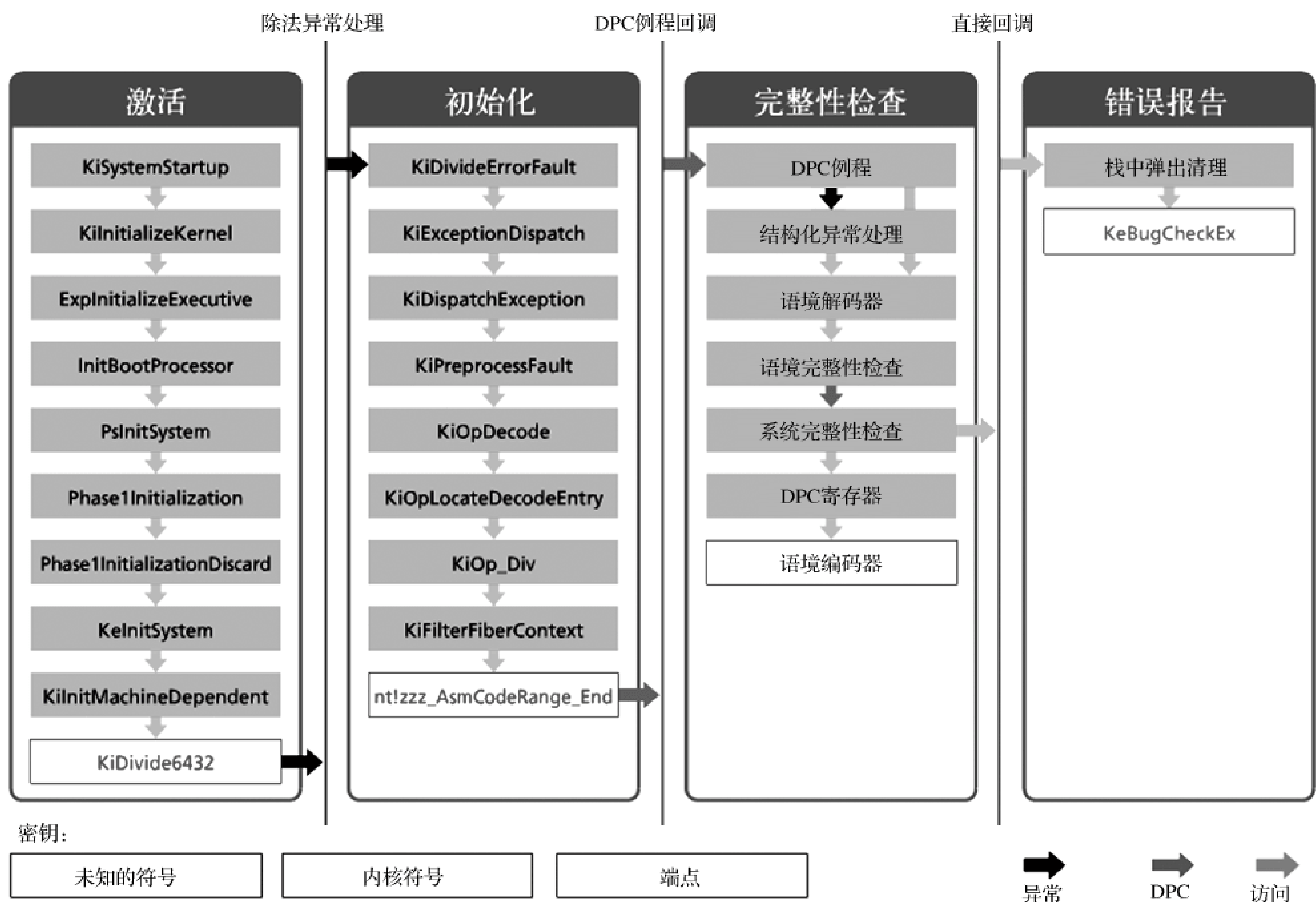


图 14－16　Patch Guard 初始化及操作过程视图

Patch Guard 是以内核驱动程序的形态存在的，且为了使 Patch Guard 机制本身免受攻击，Windows 采用了许多黑客的做法来增加定位和分析 Patch Guard 的难度，例如命名模糊得令人摸不着头脑的初始化函数（KiDivide6432、zzz_AsmCodeRange_End 等），内部也使用了很多错误来引导代码逆向者走入歧途。另外，如果系统处于调试状态，Patch Guard 机制会被系统禁用，以防对 Patch Guard 机制的恶意调试和代码逆向。

14.8　Safe Unlinking 机制

Safe Unlinking 是一种防止堆溢出的保护机制，也是从 Windows XP 系统就开始支持的安全机制。在系统中，堆是由堆分配器采用双向链表的方式维护的，例如图 14－17 的块首结构中含

有的前向指针和后向指针。当堆分配器从空闲链表中移除(Unlink)堆块时,Safe Unlinking 机制会验证 Flink 和 Blink 两个指针(前向指针和后向指针)的正确性,即是否满足 Entry→Flink→Blink = = Entry→Blink→Flink = = Entry,以防止攻击者使 Flink 或 Blink 指向任意内存地址,进而消除在执行 Unlink 操作时写入任意 4 字节数据的机会。

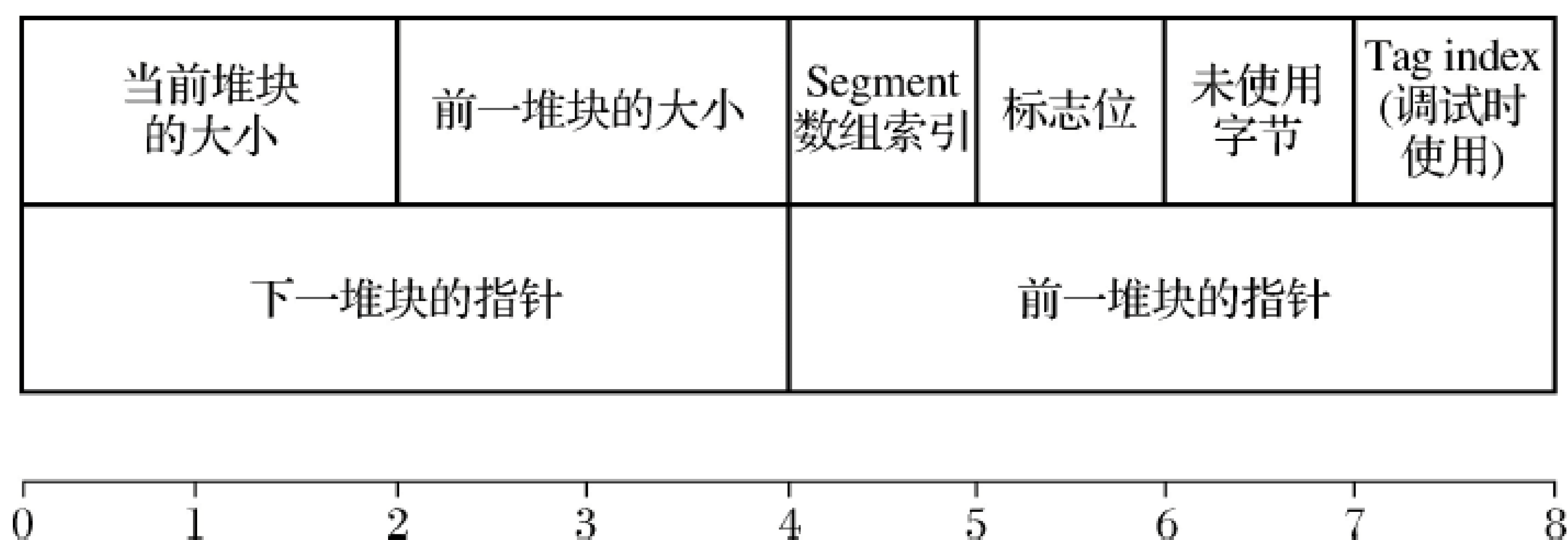


图 14 - 17　Windows 堆块块首结构

14.9　CFG 机制

CFG(Control Flow Guard,控制流保护)是 Windows 8.1 和 Windows 10 系统才支持的机制,用于保护在汇编层的直接调用。CFG 机制通过在间接跳转代码前插入校验代码以检查目标地址的有效性,进而阻止执行指令跳转到预期之外的地点,及时有效地进行异常处理,避免引发相关的安全问题。简单地说,就是在程序间接跳转之前判断这个将要跳转的地址是否合法。当然,CFG 机制的实现有赖于编译器、操作系统和用户态依赖库等的共同协作。

CFG 机制关注和缓解的是间接调用的不可靠目标地址问题。不可靠目标地址有个明显特征,即大部分情况下目标不是一个有效的函数起始地址,因此 CFG 机制是基于这样一个基本条件:间接调用目标地址必须是一个可靠的函数起始地址。

如图 14 - 18 所示,左边是未启用 CFG 机制的汇编代码,右边是启用了 CFG 机制的汇编代码。启用 CFG 机制时,在间接调用前,目标地址被传给了_guard_check_icall 这个公共函数,该函数实现了 CFG 机制;而在没有 CFG 机制支持的 Windows 系统中,_guard_check_icall 函数不做任何事。_guard_check_icall 函数实际指向 ntdll.dll 中的 LdrpValidateUserCallTarget,后者会实质判断传入的目标地址是否是一个合法的函数地址。

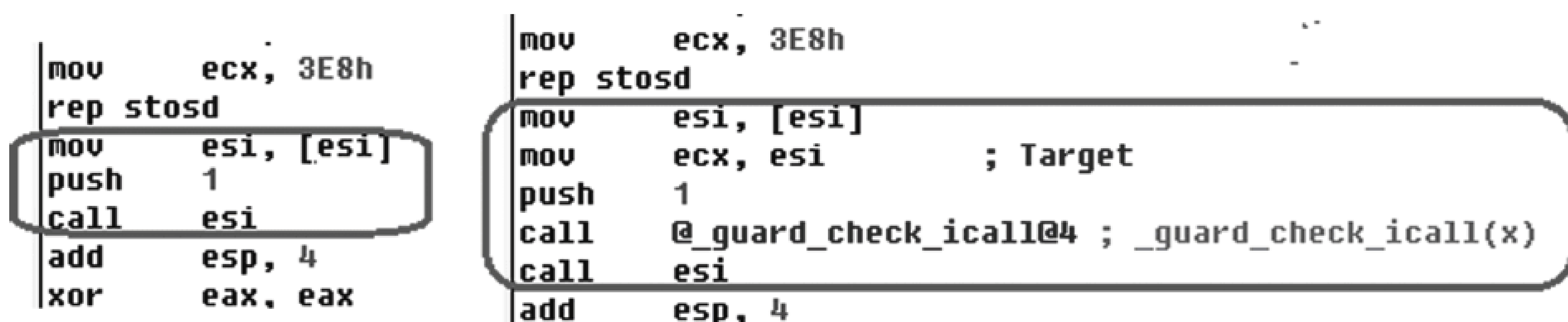


图 14 - 18　未启用 CFG 机制与启用 CFG 机制的汇编代码对比

14.10 Secure Boot 机制

从 Windows 8 开始操作系统的启动方式发生了很大的变化，即从原先 Windows 7 的 Legacy 启动方式（传统启动方式）转变为 Secure Boot 启动方式，如图 14－19 所示。已经预装了 Windows 8 系统的计算机要退装 Windows 7 也必须修改 UEFI 以取消 Secure Boot，可见 Secure Boot 机制已经硬化到计算机 BIOS 固件之中了。

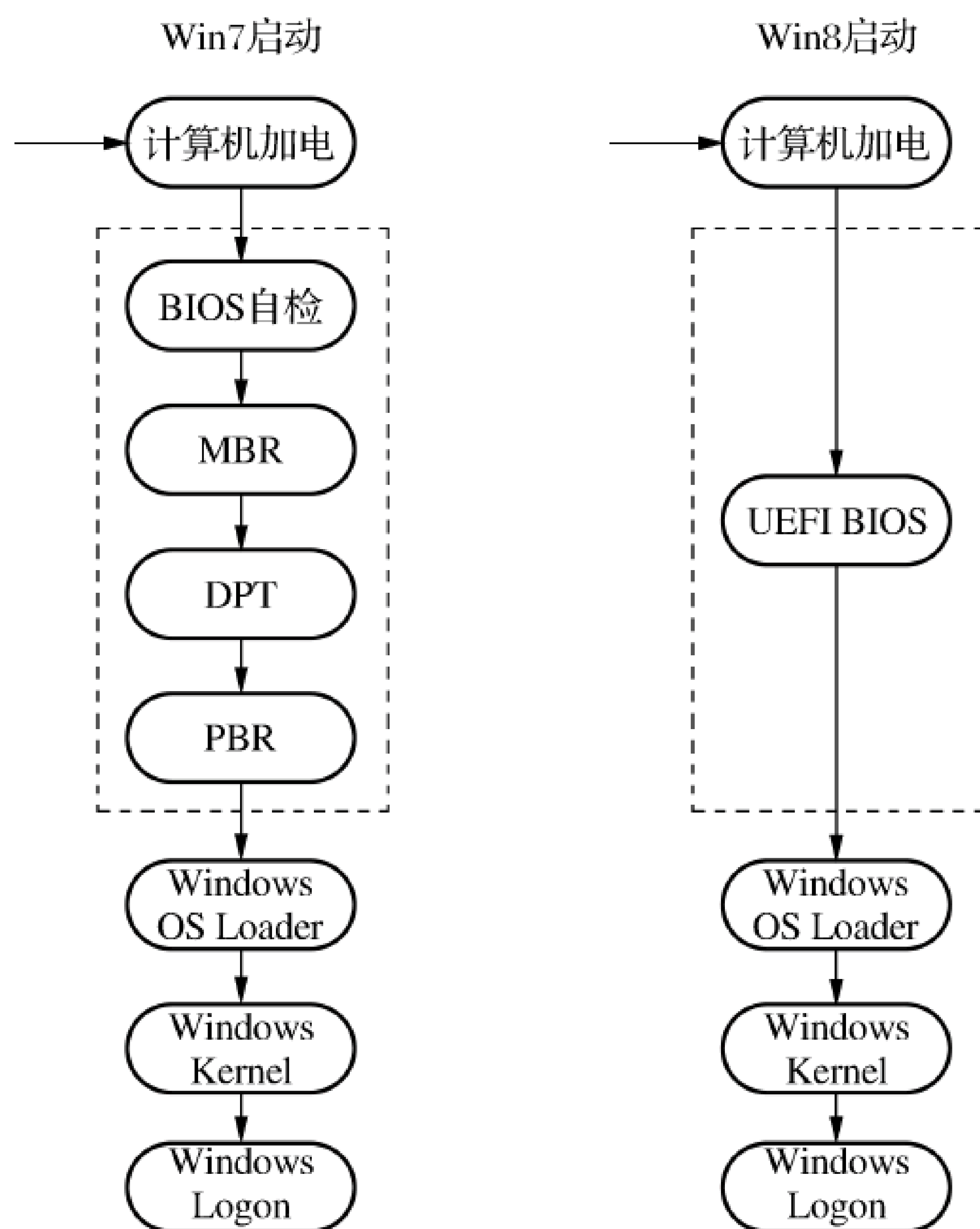


图 14－19　Windows 7 与 Windows 8 系统启动过程的对比

Secure Boot 是 UEFI 的一个子规范，也就是 UEFI 规范的一个子集，其目的是通过非对称加解密的方式防止恶意软件侵入以加强系统自身的安全性，其核心原理契合了可信计算度量的思想。目前微软已经要求主板厂商在 BIOS 中内置 Windows 8 的公钥。

Secure Boot 机制的技术原理是这样的：

- UEFI 规范规定主板出厂的时候可以内置一些可靠的公钥。
- 任何想要在当前主板上加载的操作系统和硬件驱动必须通过上述可靠公钥的认证，否则拒绝加载，即操作系统和硬件驱动必须采用公钥对应的私钥签名，从而使恶意软件无法通过验证也就不可能感染系统的 Boot 阶段。

因此，Secure Boot 机制就是对操作系统和硬件驱动的签名验签过程，操作系统和硬件驱动相当于签名过程中的发送端，而 Secure Boot 则是验签过程中的接收端，如图 14－20 所示。这也是驱动签名强制（Driver Signature Enforcement，DSE）机制的由来。

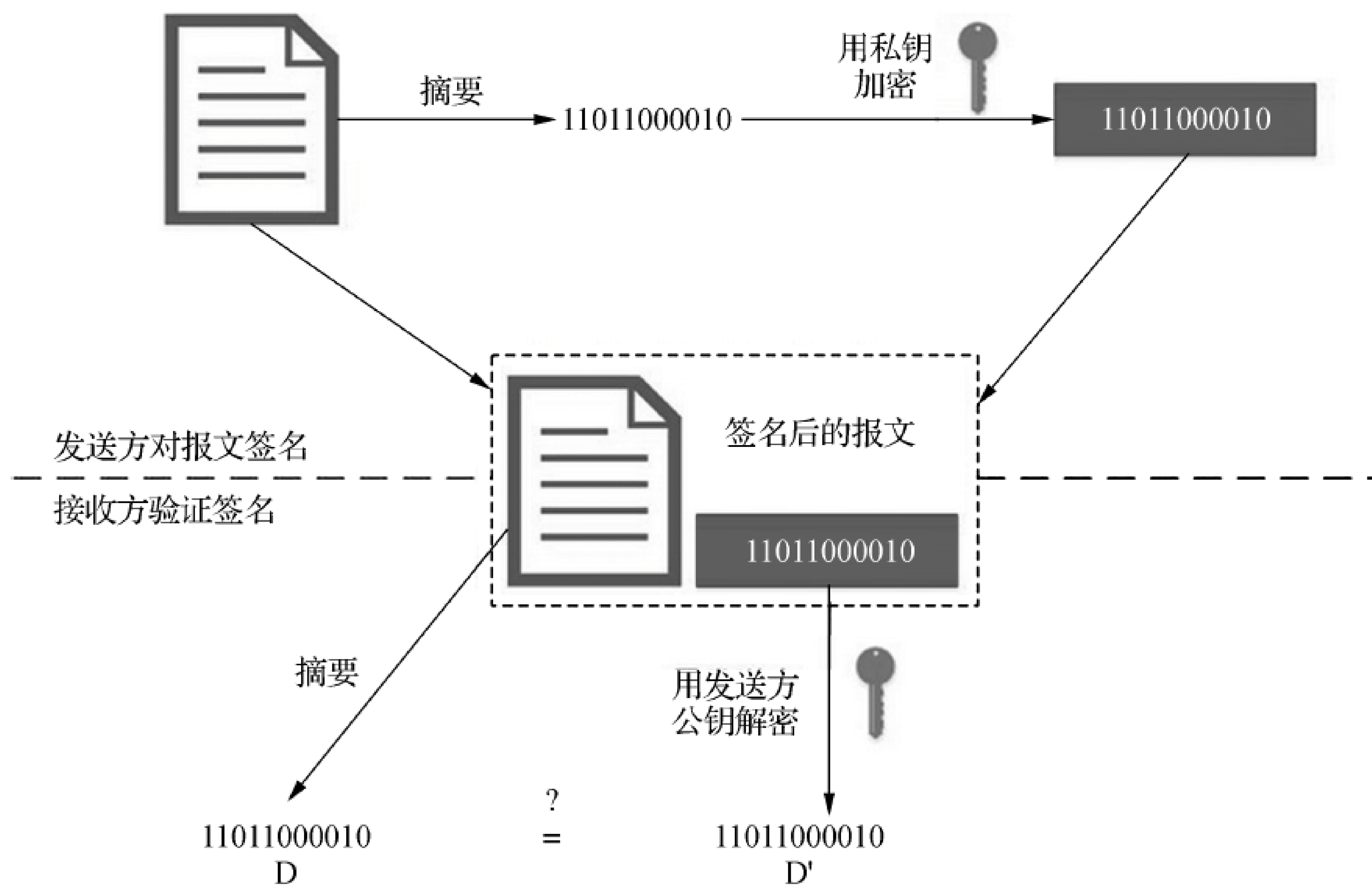


图 14－20　签名与验签过程(图片来自 CSDN)

本章小结

本章介绍了 Windows 的内存安全机制。

Windows 攻击的很大一部分为内存攻击,即由于软件/硬件漏洞而造成的缓冲区内存溢出、整数溢出等,因此杜绝内存中的各种漏洞是当务之急。在本章中按照不同的维度分别介绍了数据执行保护、加载地址随机、Security Cookie 等机制,对应解决了内存页执行、进程模块默认加载地址、堆栈溢出保护和校验等问题。

第15章　可信计算技术

可信计算是一种历史比较悠久的技术,它以加解密相关技术为基础,并基于信任链的思想对计算机系统中的软硬件进行可信性度量验证。

中国的可信计算发源于1992年立项的可信计算综合安全防护系统——智能安全卡,并逐步形成了自主的安全可信体系,近期推出的等保2.0标准(网络安全等级保护制度2.0国家标准)更是直接将基于可信计算技术的可信验证作为了三大特点之一。而从技术发展的代差来讲,可信计算技术可以分为三代:

- **可信计算1.0**:主要为了解决主机系统可靠性问题,面向的是计算机系统中的各个部件,技术手段也是比较原始的基于容错算法的故障诊查方式。
- **可信计算2.0**:主要为了解决节点安全性问题,面向的是PC单机,技术手段是基于TPM+TSS的被动度量方式。
- **可信计算3.0**:这是目前最为先进的技术方式,主要为了解决网络安全性问题,采用公私钥主动免疫技术对终端、服务器、存储系统和网络进行可靠性验证,其主要技术特点是主动免疫、动态度量、实时感知。可信计算3.0是由中国首先提出的具有极大创新性的安全技术,利用可信计算3.0技术可以有效排除"熔断"、"幽灵"等CPU固件漏洞。

本章节将按照图15-1所示的提纲对可信计算技术进行梳理。

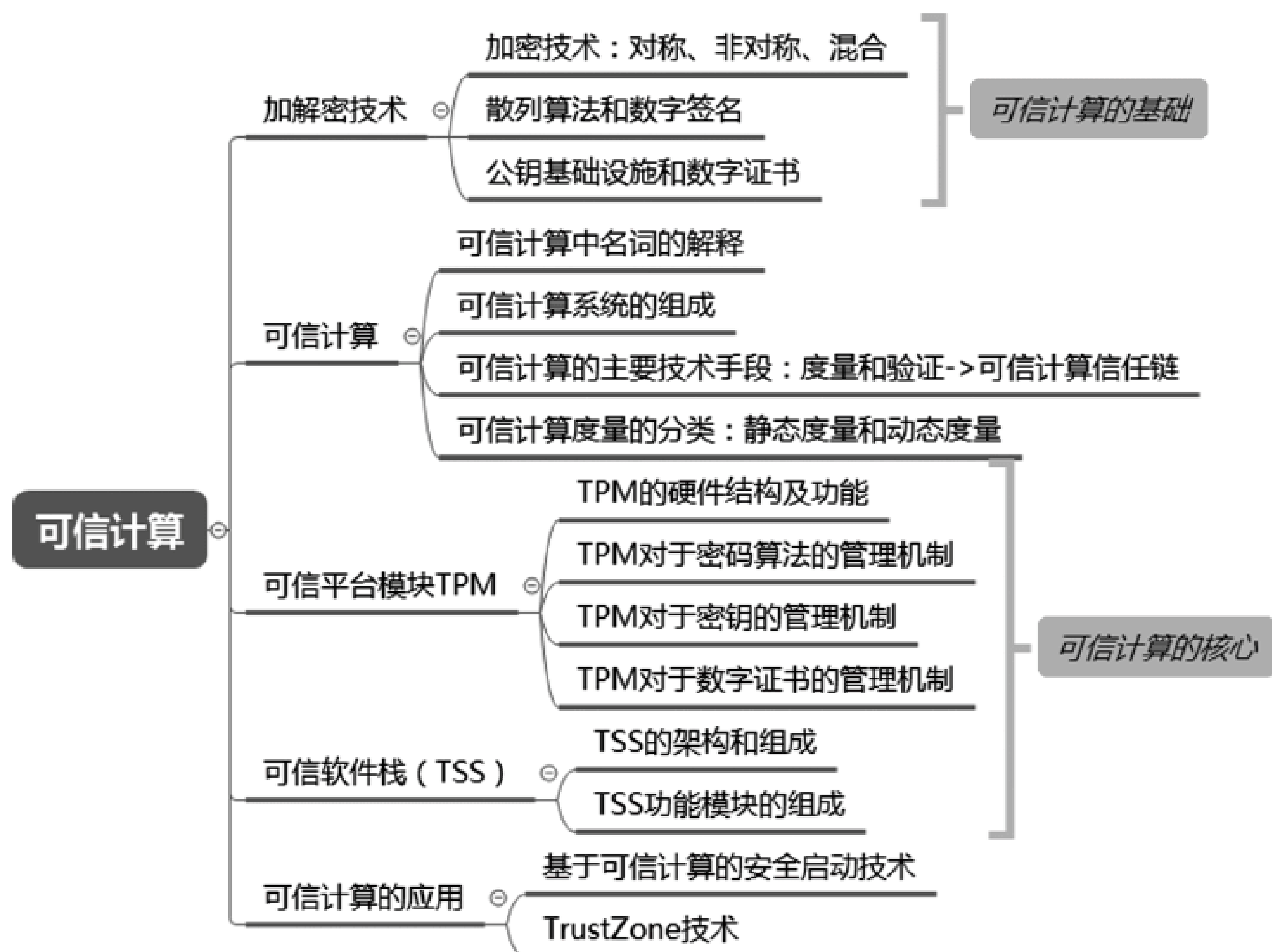


图15-1　本章提纲

15.1　加解密技术

1. 加密技术

在介绍可信计算技术之前,我们必须首先介绍加解密技术,这是因为加解密技术是可信计算的基础技术。可按照加解密密钥是否一致将加密技术分为以下几类:

- **对称加密技术**:即采用单钥的加密方法,加密与解密使用同一个密钥,如图 15-2 所示。因为不分公钥与私钥,因此被称为对称加密技术。

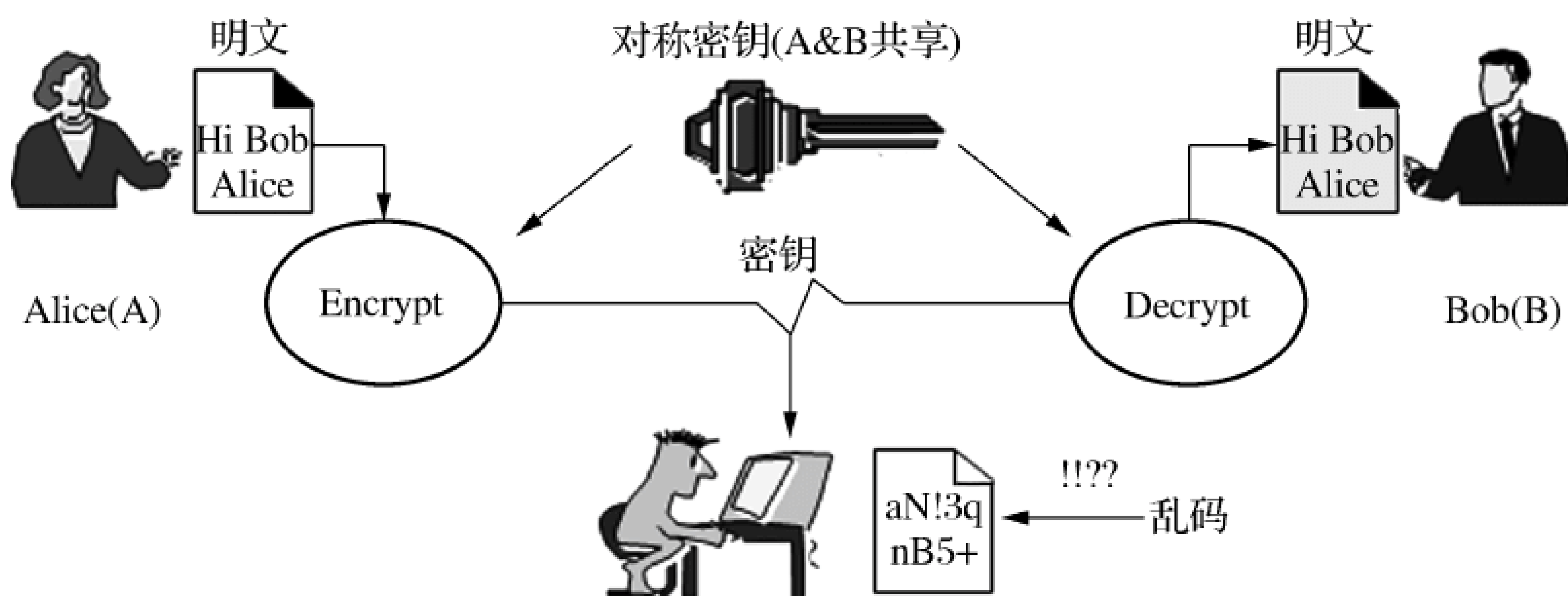


图 15-2　对称加密技术在数据传输中的应用场景(图片来自 CSDN)

- **非对称加密技术**:公钥与私钥是一对不同的密钥,如果用公钥对数据进行加密,则只有对应的私钥才能解密,如图 15-3 所示;如果用私钥加密数据,则使用公钥才能解密,如图 15-4 所示。因为加密和解密密钥不同,所以这种方法叫作非对称加密技术,可用于身份认证等场景,如图 15-5 所示。

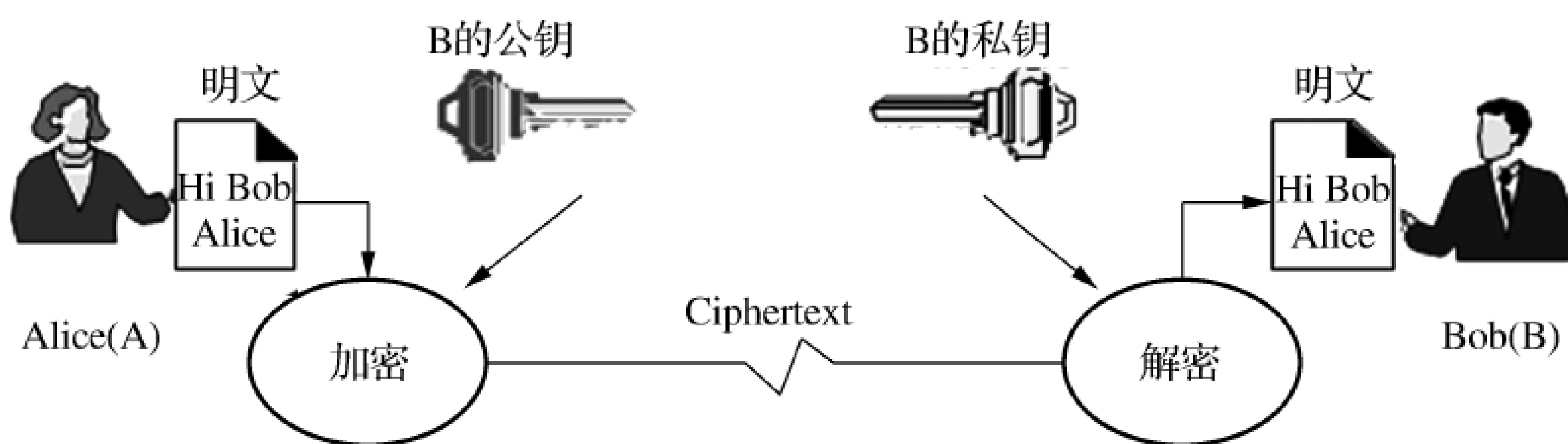


图 15-3　公钥加密、私钥解密的数据加密场景(图片来自 CSDN)

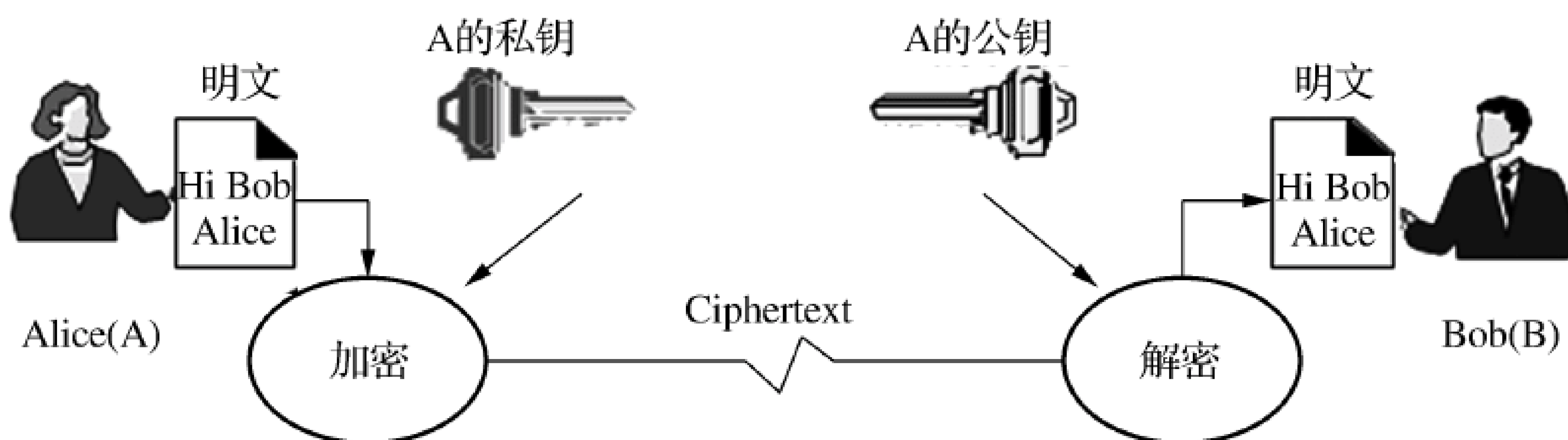


图 15-4　私钥加密、公钥解密的数据加密场景(图片来自 CSDN)

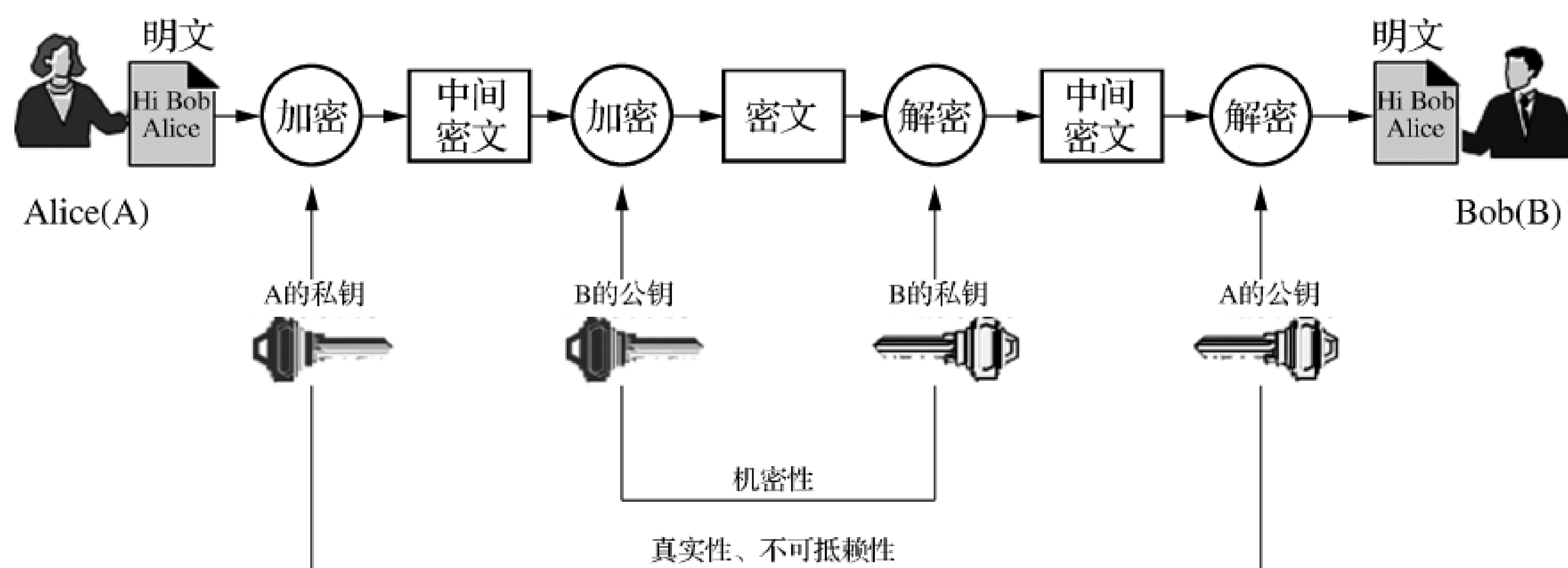


图 15-5　身份验证的密文在数据传输过程中的加解密(图片来自 CSDN)

- **混合加密技术**:用非对称算法交换对称密钥,用对称密钥加密数据,这种方法称为混合加密技术,如图 15-6 所示。混合加密技术不能实现身份验证。

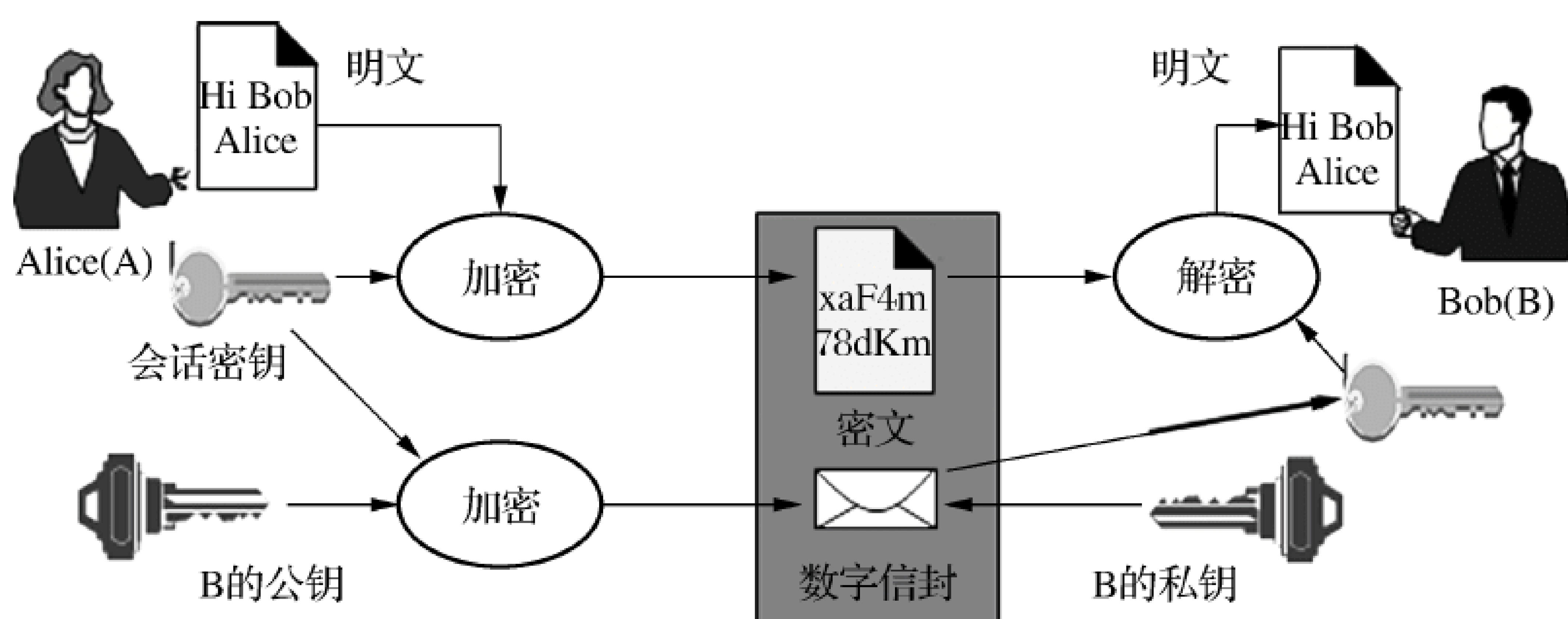


图 15-6　混合加密技术在数据传输中的应用场景(图片来自 CSDN)

对于对称加密技术,常见的加密算法有:DES(数据加密标准)、3DES、IDEA(国际数据加密算法)、Blowfish、RC4、RC5、Rijndael、AES(高级加密标准)等。

对于非对称加密技术,公钥常用于数据加密(用公钥加密)或签名验证(用公钥解密),私钥常用于数据解密(发送端用接收端的公钥加密)或数字签名(用自己的私钥加密),常见的加密算法有 Diffie-Hellman、RSA、EI Gamal、椭圆曲线、DSA、背包、DSS 等。

2. 散列算法

所谓**散列(Hash)算法**,也被称为哈希算法,输入任何长度的消息,通过一个单向的运算产生定长的输出,这个输出的值被称为散列值,并且散列算法是不可逆的。散列算法的种类包括 MD2、MD4、MD5、HAVAL、SHA、Tiger、RIPEMD-160 等。散列算法有多重应用,例如通过摘要的方式验证数据是否被篡改,其步骤如下:

(1) 报文发送端使用散列算法对即将发送的报文计算消息摘要(Abstract1)。

(2) 发送端将 Abstract1 附在发送的报文明文之后并发送给接收端。

(3) 接收端接收到报文后使用相同的散列算法计算出报文明文的消息摘要(Abstract2)。

(4) 将 Abstract1 与 Abstract2 相比较,如果相同则表示消息报文未被篡改。

当然，上述过程不能实现真正的完整性保证，因为中间人可以在截获消息报文后按照约定的散列算法重新计算消息摘要并附在报文之后，这样就可以蒙蔽接收端。针对这种漏洞可以采用数字签名技术来解决。

所谓数字签名，就是用自己的私钥对原始数据的消息摘要进行加密后得到的数据，也就是先散列再加密，消息摘要是作为加密端输入参数的，如图 15－7 所示。

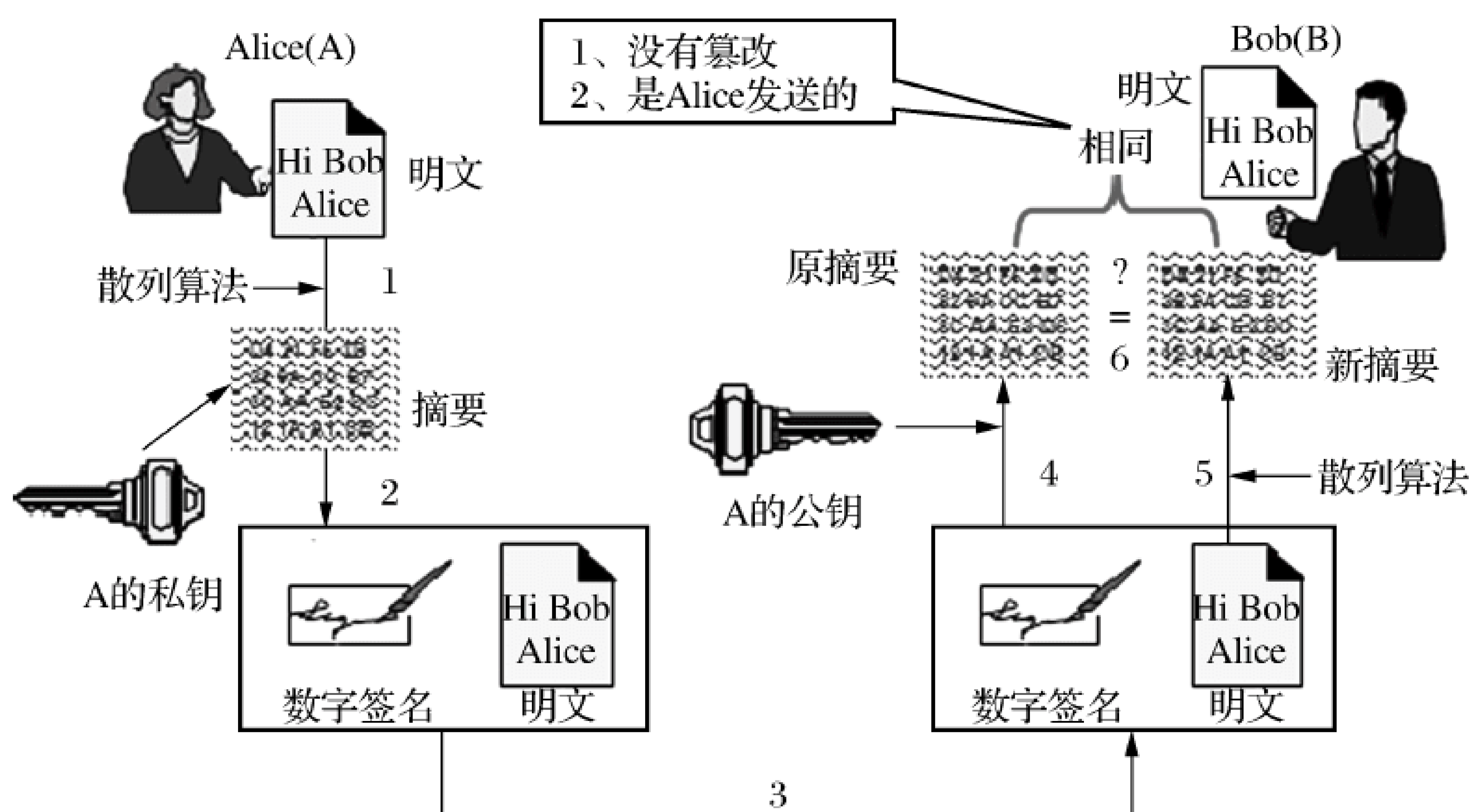


图 15－7　数字签名的生成和解析过程(图片来自 CSDN)

由于对消息摘要做了私钥加密，因此中间人无法对截取的摘要内容进行解密和散列（无法知道加密算法），从而保证了消息的可靠性。

3. 公钥基础设施

公钥基础设施（Public Key Infrastructure，PKI）是利用公钥概念和加密技术为网络通信提供的符合标准的一整套安全基础平台，也是一套由软件、通信协议、数据格式、安全策略等组成的用于使用、管理、控制公钥密码机制的系统。公钥基础设施有三个主要功能：

- 发布公钥和证书；
- 证明绑定公钥的实体；
- 提供公钥的有效性验证。

PKI 技术采用数字证书管理公钥，通过第三方可信任的认证中心（Certificate Authority，CA），把用户的公钥和其他标识信息（如名称、E-mail 地址、身份证号等）捆绑在一起，在互联网上验证用户的身份。数字证书的使用过程如下：

（1）发送者生成一个密钥对（公钥 + 私钥），并把私钥保存起来，用公钥向 CA 申请证书。

（2）CA 接受申请，给发送者颁发数字证书，证书中包含发送者的公钥和其他身份信息。

（3）同时 CA 会计算这些信息的消息摘要并用自己的私钥加密消息摘要（数字签名）并附在发送者的证书上，以此来证明这个证书就是 CA 自己颁发的。

（4）接收者得到上述通过 CA 加持过的证书后用 CA 证书中的公钥来解密消息摘要。

(5) 通过步骤(4)可以确认发送者的证书是 CA 颁发的,证书未被篡改,同时也得到了发送者的公钥。

15.2 可信计算

可信计算是由可信计算组织(Trusted Computing Group,TCG)推动和开发的安全计算技术,其基本原理是在计算和通信领域中广泛使用基于硬件安全模块的可信计算平台以提高整个系统和应用软件的安全性与完整性。作为可信计算的技术推动和标准制定组织,TCG 由 AMD、惠普、IBM、英特尔和微软于 2003 年组成,并逐步取代了 1999 年成立的可信计算平台联盟。目前 TCG 已经发展成拥有几百个成员的国际性组织。

1. *名词解释*

在介绍具体技术之前,我们先来解释一些概念和名词。

- **TC**:可信计算(Trusted Computing)。
- **TCB**:可信计算基(Trusted Computing Base),是计算机系统中负责执行安全策略的全体实体集合,包括硬件、软件和其他固件。其更抽象的理解就是使平台可信的最小代码集。
- **TPM**:可信平台模块(Trusted Platform Module),是具有加密功能的安全微控制器,旨在提供加密密钥等基本安全功能。TPM 芯片一般集成在主板上并通过硬件总线与系统的其他部件通信。
- **TCM**:可信密码模块(Trusted Cryptography Module),TCM 是可信计算平台的硬件模块,包括密码计算区、受保护的存储区和主计算区。TCM 为 TPM 提供密码运算功能,包括平台完整性度量、平台免疫力建立、为平台身份提供唯一标识等。
- **TSS**:可信软件栈(Trusted Software Stack),是可信计算平台的支撑软件,用来向其他软件提供安全芯片的接口,并通过安全机制增强操作系统和应用程序的安全性。
- **TNC**:可信网络连接(Trusted Network Connection),用于解决网络环境中终端主机的可信接入问题,在主机接入网络之前,必须检查其是否符合该网络的接入策略(如是否安装有特定的安全芯片和防病毒软件等)。
- **信任根**:根据功能分为可信度量根(RTM)、可信存储根(RTS)和可信报告根(RTR)。
- **RTM**:可信度量根(Root of Trust for Measurement),是一个进行完整性度量的计算引擎,以 CRTM 为核心信任代码,系统的状态和信任链信息被保存在 PCR 中以通知其他模块。
- **CRTM**:核心信任根模块(Core Root of Trusted Module),作为可信度量根(RTM)的核心代码,CRTM 在可信计算中被认为是绝对可信的代码模块,是计算机系统可信的基点,也是系统启动执行的第一段代码。CRTM 对计算平台的可信性进行度量,对度量的可信性进行存储,并提供可信性报告。CRTM 可以是一段 BIOS 引导程序,也可以

是磁盘或芯片上的一段代码。

- **RTS**：可信存储根（Root of Trust for Storage），一般是 TPM 芯片中的一组 PCR 和存储器，记录了完整性度量的摘要值。RTS 将完整性度量保存在日志中，并将它们的散列值保存在 PCR 中。
- **RTR**：可信报告根（Root of Trust for Report），是一个 RTS 可靠报告的计算引擎。RTS 保存委托给 TPM 的密钥和数据并管理少量的内存，其中存放的密钥用于解密和签名操作。
- **TBB**：可信构建模块（Trusted Building Block），是根信任中不包含屏蔽区域或保护功能的部分。一般来讲，TBB 只包含用于 RTM 的指令和 TPM 初始化功能，这些指令和功能是不同平台所特有的。
- **Intel TXT**：Intel 可信执行技术。
- **密封**：密封是指 TPM 对数据块进行加密使其不能被解密的行为。

2. 可信计算系统的组成与主要技术系统

通过前面的介绍，我们在此梳理一下：可信计算系统一般由以下几部分（按照从底层到上层、从内层到外层的顺序排序）组成，如图 15－8 所示：

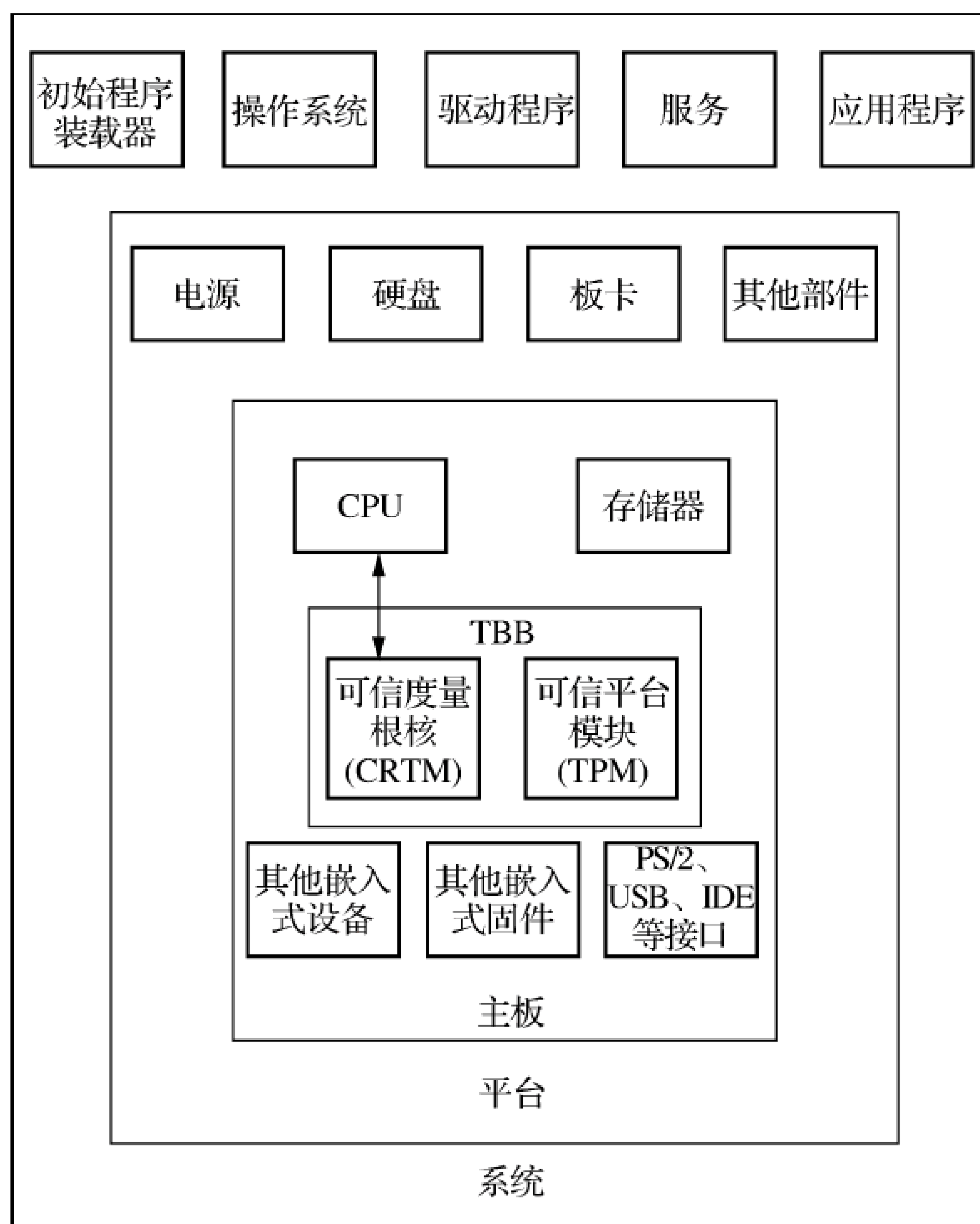


图 15－8　可信计算机系统的组成

- **最核心的模块**：TBB，一般由核心信任根模块（CRTM，也被称为可信度量根核）和可信

平台模块(TPM)构成。CRTM 负责提供最核心的可信计算基,TPM 负责基本的安全计算功能。

➢ **主板模块**:包括 CPU、Cache、内存、嵌入式设备、其他固件(BIOS 等)以及 USB 和 IDE 接口等。这部分是普通计算系统都应该具备的模块。

➢ **平台模块**:对应的外设(硬盘、电源和其他板卡等)。

➢ **软件模块**:整个系统中最上层的部分,包括操作系统加载器程序、操作系统、设备驱动、系统服务以及更上层的应用软件。

可信计算作为一门新兴学科,其主要的技术目标包括:

➢ 计算平台的完整性;

➢ 平台的远程证明;

➢ 数据存储的安全性;

➢ 数字知识产权保护。

经过这些年的发展,目前可信计算的主流技术手段是利用认证密钥对软件度量值进行签名和验签,而认证密钥和软件度量值都可通过可信根来进行安全存储。可信计算的目的也是保证整个系统软件栈的安全与完整,防止恶意篡改,这同时也是“可信”的技术内涵。当然可信计算不能百分百地杜绝完整性问题,但却可以有效预防和减少安全性问题的发生。

可信计算的主要技术手段一个是度量,一个是验证。图 15－9 展示了基于静态度量的执行流程。

➢ 所谓**度量**,就是采集和检测目标软件系统的状态,包括静态状态(磁盘文件上的二进制特征码等)和动态状态(系统调用等软件行为特征码)。

➢ 所谓**验证**,就是将度量的结果与参考值进行比对,如果不一致则表明软件系统发生了篡改。

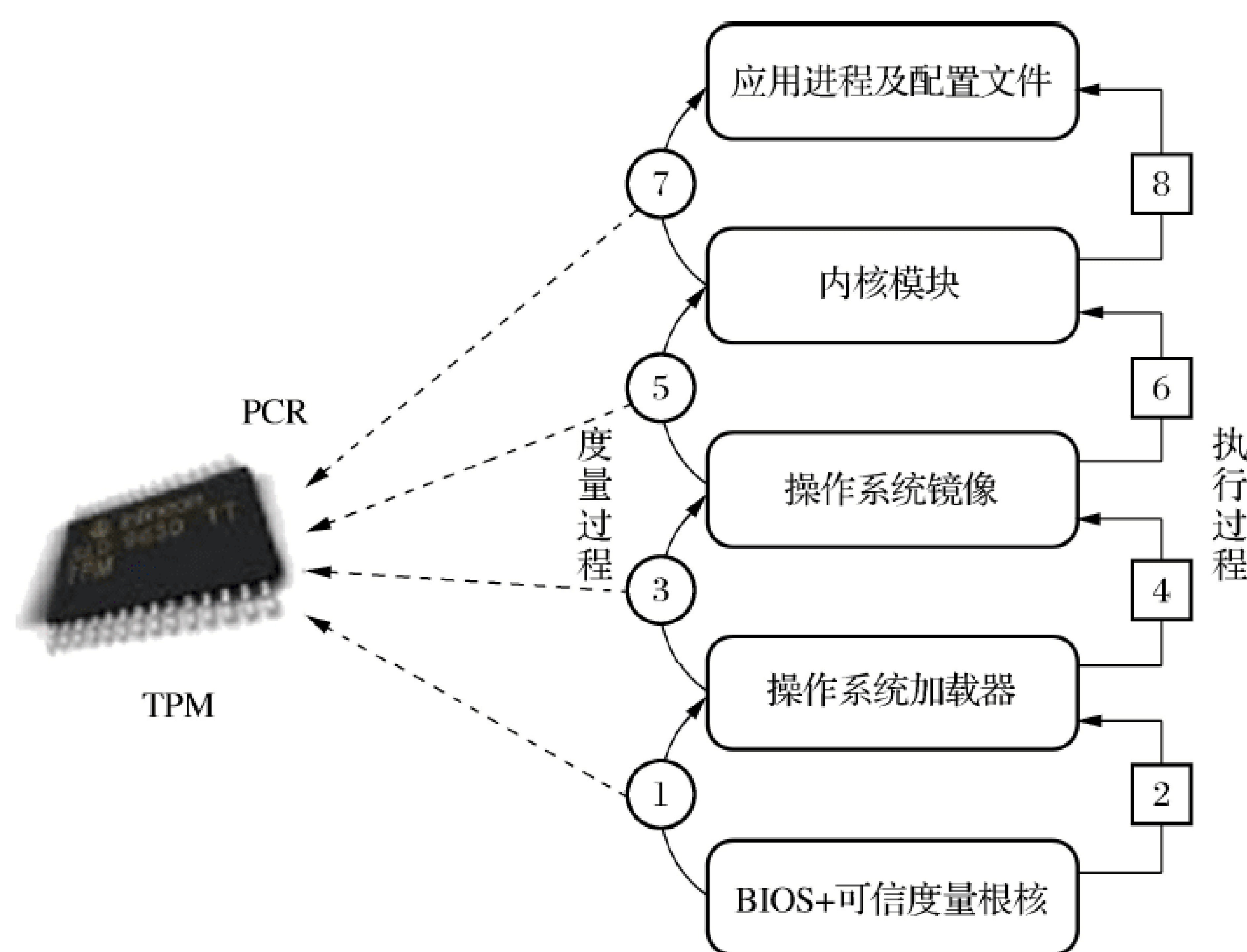


图 15－9　基于静态度量的执行流程(左边为度量过程,右边为执行过程)

度量是一级一级地从底层向上逐级进行的，通常是以先启动的软硬件的代码（例如安全芯片和 CRTM）作为信任根，并以此为基准对后一级启动的软件进行度量，以此实现信任链的向后传递，以保证系统计算环境可信。整个过程遵循“先度量再执行”的策略。例如在 Windows 系统启动时，可先以 BIOS 或 TCM 中的某段代码为核心信任根模块（可信根），对启动链上的 BIOS/UEFI、WinLoader、操作系统镜像文件等进行逐级静态度量，如图 15 - 10 所示的物理机与虚拟机的信任链逐级度量。

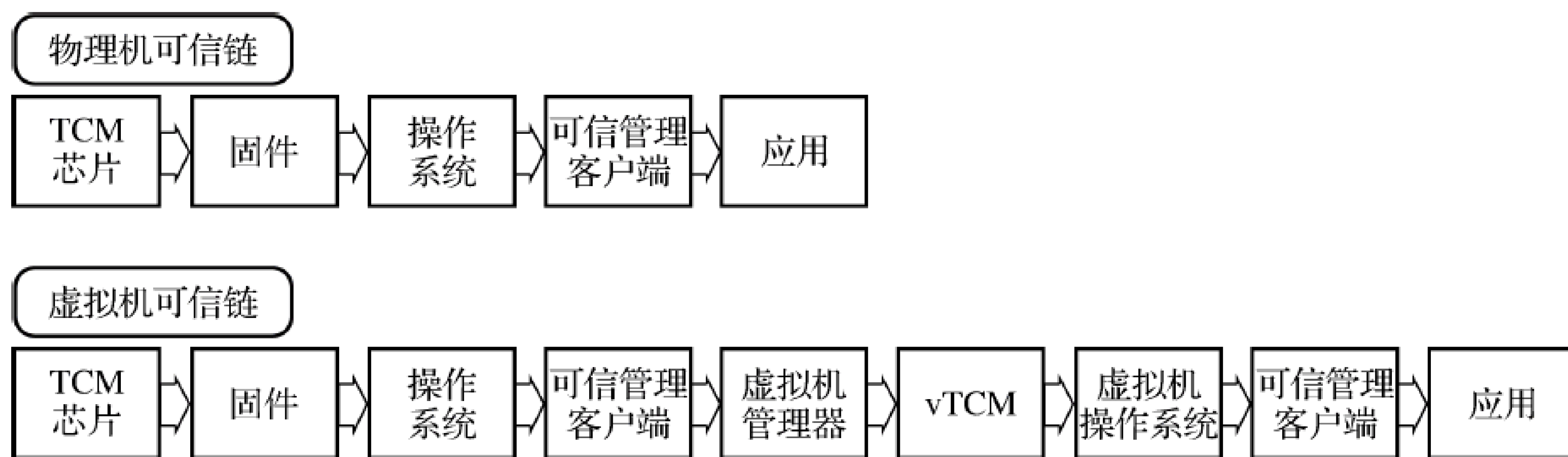


图 15 - 10　物理机与虚拟机系统启动信任链

可信根作为整个系统信任链的最底端必须绝对可信，因此可信根一般是通过厂家在安全芯片中直接植入算法和密钥实现的，具有不可覆盖性，因此这部分代码也被称为可信软件基（Trusted Software Base，TSB）。

根据安全芯片和可信软件基分类的可信计算标准有下列三类：

- **TPM**：可信平台模块（Trusted Platform Module），技术成熟，商业化时间长，生态得到微软、谷歌等公司的支持。
- **TCM**：可信密码模块（Trusted Cryptography Module），与 TPM 一样，技术成熟，生态完善。
- **TPCM**：可信平台控制模块（Trusted Platform Control Module），是国产化的可信标准，对硬件和 TSS 架构做了很大调整，支持主动度量，但是技术和产品均不成熟，生态也不完善。

可信计算作为一门安全科学，其涵盖的领域很大，一般来说包括计算环境可信、网络环境可信、网络接入可信、数据存储可信等。可信性度量手段亦包括静态与动态两种：在系统启动时采用静态度量手段，在系统运行时可采用动态度量手段以检测执行环境的可信性，而可信计算的一般性信任链的构成如图 15 - 11 所示。

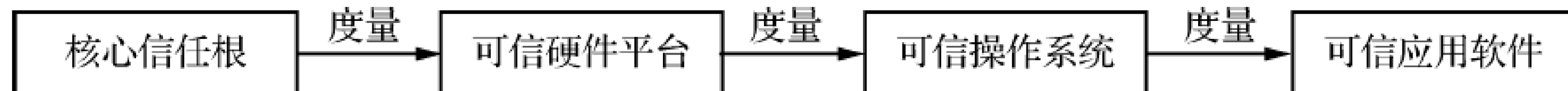


图 15 - 11　可信计算的信任链构成

目前中国等保 2.0 标准的四级安全已要求应用软件的可信性能够动态验证，对行为异常的应用进程能够进行阻断、删除甚至重启系统等。特别是基于云平台的可信计算，对于包括计算固件、虚拟机管理器、Host OS、Guest OS、容器、应用进程等在内的一系列实体的可信

性，怎样度量和验证又是一个崭新的课题。

不过基于信任链的可信计算也有自己的不足，包括：

➢ 信任链越长，信任的损失越大；

➢ 信任链的维护比较麻烦，牵一发而动全身；

➢ CRTM 存储于 TPM 之外，易遭受篡改攻击等。

后面几节我们会分别介绍 TPM、可信软件栈、可信计算的应用。

3. 动态可信性度量

动态可信性度量主要基于将软件执行过程中的行为与软件的预期行为进行比对，即通过行为认证码的方式在各个监控点上监控行为是否可信。监控的对象是数据指纹和行为指纹，**数据指纹**包括数字签名、摘要等；**行为指纹**包括系统行为（如挂钩行为、模块注入等高危动作）、进程行为（如进程注入等）、网络行为（如非法外联等）和文件操作等，如图 15－12 所示。

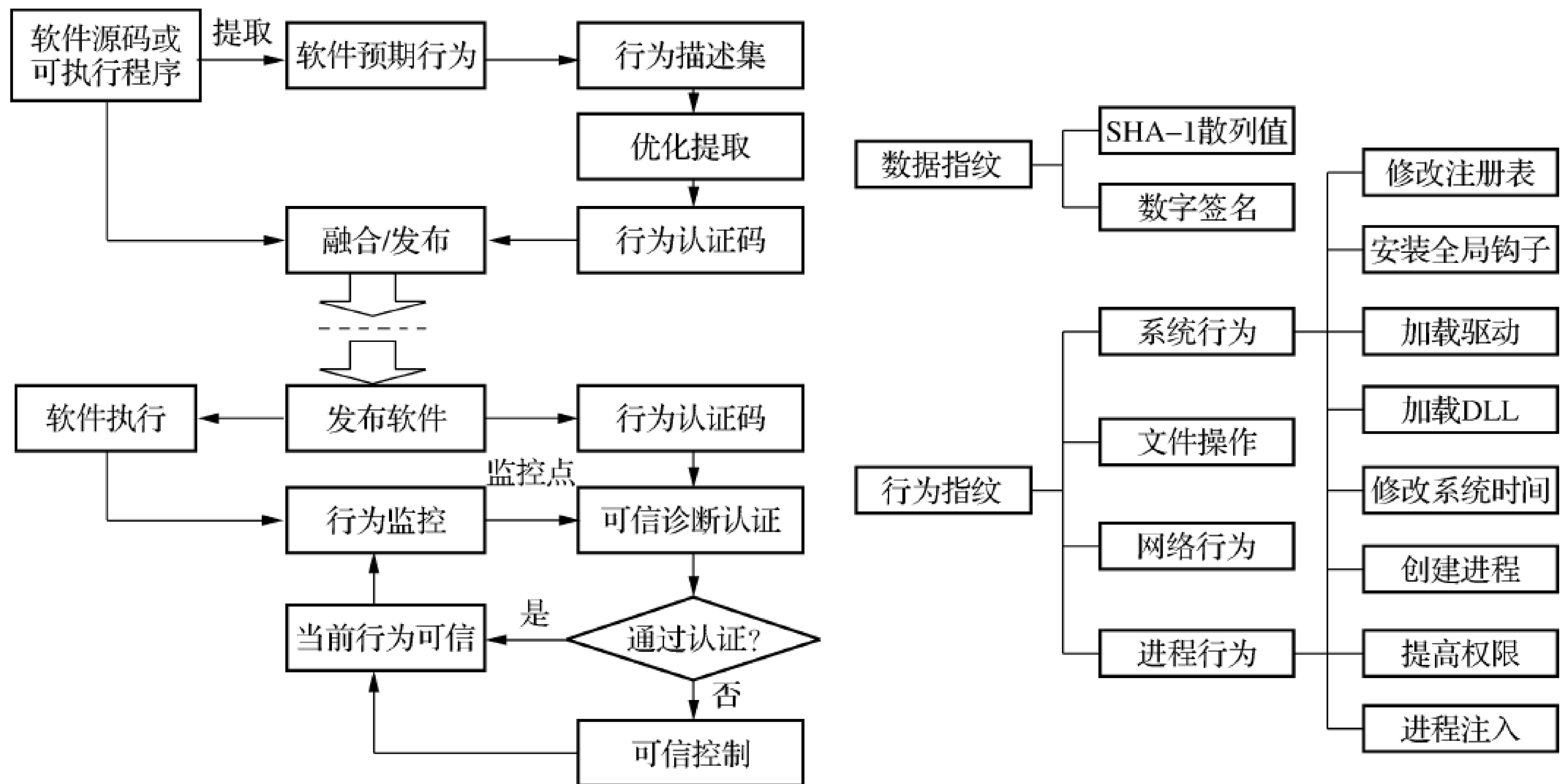


图 15－12　动态可信性度量的基本思想与数据指纹和行为指纹

15.3　可信平台模块

1. TPM 的结构

TPM（可信平台模块）作为可信计算平台的信任根是一种片上系统（System On Chip，SOC），是含有密码运算部件和存储部件的系统级芯片，是系统的核心。TPM 被集成到主板上，通过总线与外部系统交互。因此 TPM 的上层是可信软件栈（TSS），TSS 的上层就是可信平台的应用软件了。每个厂家对于 TPM 的结构都有不同的定义，TCG 定义的 TPM 由执行引擎、密码协处理器、密钥产生部件、随机数产生部件、摘要生成部件、消息认证码引擎、电源检

测部件、配置开关、易失性存储器、非易失性存储器和 I/O 部件等构成，其结构如图 15－13 所示，具体如下：

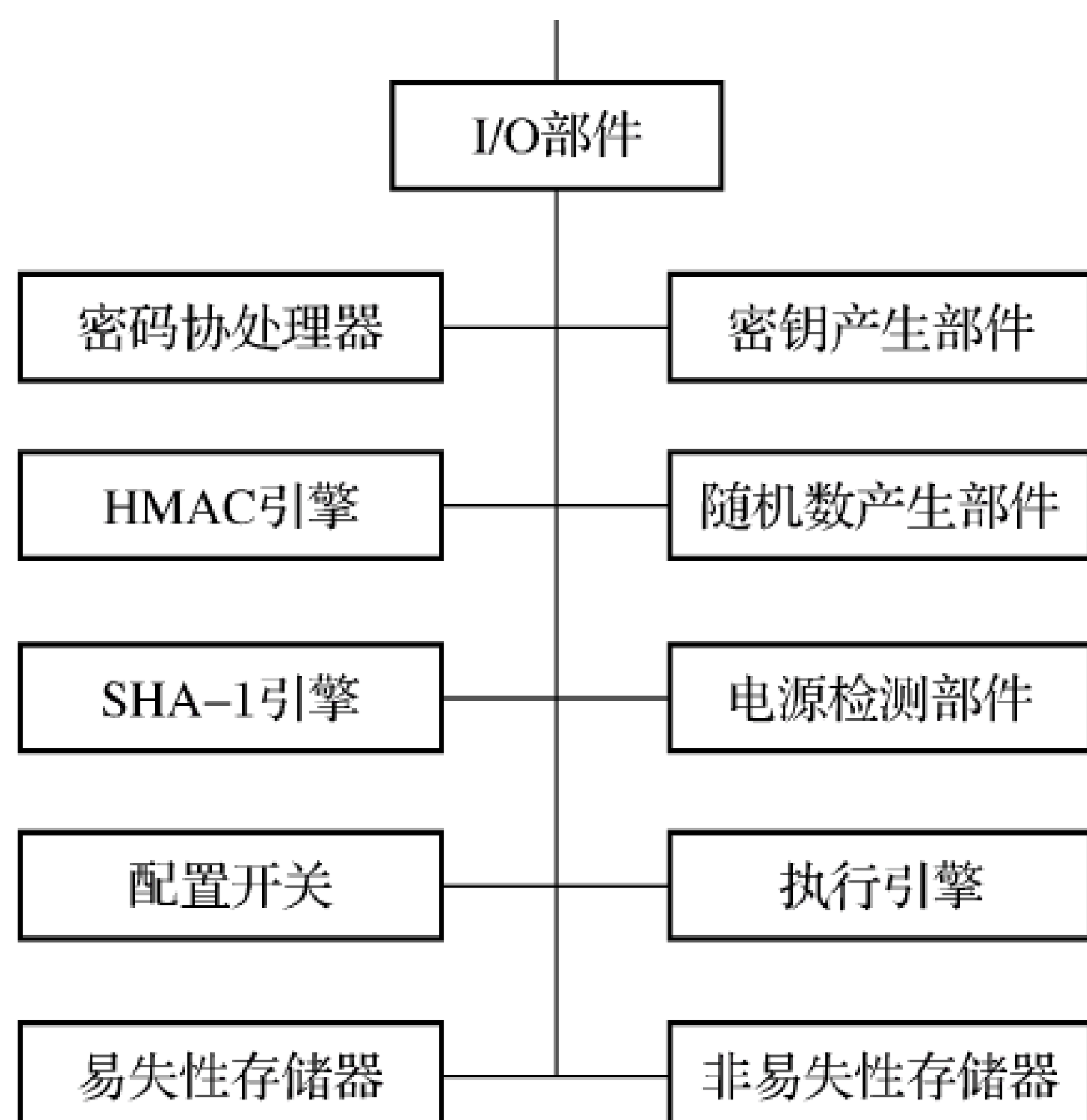


图 15－13　TCG 定义的 TPM 结构

这些部件的作用或组成如下：

- **执行引擎**：CPU 以及相应固件。
- **密码协处理器**：加解密、签名和验证签名的硬件加速芯片，采用 RSA 算法，也允许使用 ECC 或 DSA 算法。
- **密钥产生部件**：产生 RSA 密钥对，用于数字签名和加密。
- **随机数产生部件**：TPM 的随机源，产生随机数和对称密码的密钥。
- **SHA-1 引擎**：是基于散列算法 SHA1 的硬件引擎，这种散列算法产生 20 字节的摘要，用于授权和身份验证。
- **HMAC 引擎**：是基于散列算法 SHA1 的消息认证码硬件引擎。
- **电源检测部件**：监视 TPM 的电源状态。
- **配置开关**：对 TPM 的资源和状态进行配置。
- **非易失性存储器**：存储密钥标识等重要数据。
- **易失性存储器**：TPM 的工作存储器。
- **I/O 部件**：负责 TPM 的对内对外通信。

TPM 的功能包括对 CPU 的数据流进行加密、监测系统底层状态等。由于 TPM 能够生成加密密钥、存储密钥并进行身份认证，因此利用 TPM 芯片，可以开发身份识别、系统登录加密、数据文件加解密、网络通信加解密等应用服务。不过 TPM 的设计也决定了其不适合大数据量的加解密吞吐，且其主要是面向 PC 而非服务器和移动端。

TPM 也可被看作一套规范，这套规范定义了安全密码处理器的规格，或可被看作这套规范的实现（例如上文中的 TPM 结构）。TCG 在 2000 年之后已经建立起了比较完善的 TPM1.2 技术规范体系，包括：

- TPM 功能与实现规范(TPM Main Specification Version 1.2);
- TSS 功能与实现规范(TSS Specification Version 1.2);
- 针对 PC 平台的 TCG 规范(TCG PC Client Specific Implementation Specification for Conventional BIOS Version 1.2);
- 针对服务器平台的 TCG 规范(TCG Server Specific Implementation Specification for TCG Version 1.2 和 TCG IPF Architecture Server Specification);
- 基础设施技术规范,包括身份证书、网络认证协议、完整性收集和完整性服务等。
- 针对外设在可信计算架构中的技术规范;
- 针对网络安全存储的 TCG 规范(TCG Storage Architecture Core Specification 1.0)。

2. TPM 中的密码算法管理

按照 TPM 版本的不同,TPM 对于密码算法的管理分为以下两个版本:

- **TPM1.2**:该版本的密码算法支持 RSA 加密与签名、RSA-DAA、SHA1、HMAC。TPM1.2 并未要求必须支持对称加密算法。
- **TPM2.0**:该版本的密码算法支持 RSA 加密与签名、ECC 加密与签名、ECC-DAA、ECDH、SHA1、SHA256、HMAC、AES 等,TPM 芯片厂商可以随意增加新的算法,例如国密系列的 SM2、SM3 和 SM4 等,具有一定的扩展性。

表 15-1 列出了目前常见的加密算法、签名算法和散列算法的名称与解释。

表 15-1　常见密码算法

密码算法	算法名称	备注	分类
RSA	Rivest-Shamir Adelman 算法	非对称加密算法,且两个密钥都可以用于加密(通常用公钥加密,用私钥解密)。RSA 算法可用于加密和签名,加密或签名后的结果不可读。签名的场景下,用私钥签名,用公钥验签。由于算法效率问题,一般用 RSA 算法加密比较短的对称密码	加密和签名算法
DES	数据加密标准(Data Encryption Standard)	最为流行的对称加密算法,同时也是一种分组(每次处理固定长度的数据段)加密算法,分组大小为 64 位	加密和签名算法
DSA	数字签名算法(Digital Signature Algorithm)	用于数字签名时签名生成速度很快,验证速度很慢,加密时更慢,但解密时速度很快,安全性与 RSA 密钥相等,密钥长度也相等	加密和签名算法
ECDSA	椭圆曲线数字签名算法	DSA 算法在椭圆曲线上的模拟,计算参数小、密钥短、安全强度高、运算速度快、签名短小,适用于计算与存储资源受限的场景	加密和签名算法
DAA	直接匿名认证(Direct Anonymous Attestation)	可用于可信计算平台身份认证,比较流行的是基于椭圆曲线和双线性映射的 DAA 方案(ECC-DAA)	身份认证算法
SHA1	安全散列算法(Secure Hash Algorithm)	适用于数字签名标准里面定义的数字签名和摘要算法,摘要输出为 160 位	散列算法

续表 15－1

密码算法	算法名称	备注	分类
SHA256	安全散列算法	摘要输出为 256 位，又称为 SHA2	散列算法
SHA512	安全散列算法	摘要输出为 512 位	散列算法
MD5	消息摘要算法	一种被广泛使用的密码散列函数，摘要输出为 128 位	散列算法
ECC	椭圆曲线加密（Elliptic Curve Cryptography）	一种公钥加密体制	加密和签名算法
ECDH	椭圆曲线迪菲－赫尔曼秘钥交换（Elliptic Curve Diffie-Hellman Key Exchange）	一种密钥协商算法而非加密算法，可以使用该算法在公网通道上进行安全的密钥分派	密钥协商算法
HMAC	密钥相关的散列运算消息认证码	HMAC 运算利用散列算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出	散列算法
AES	高级加密标准（Advanced Encryption Standard）	一种常见的对称加密算法	加密和签名算法
SM1	商密 1 号算法	国家密码管理局编制的一种商用密码分组标准对称算法，也是分组密码算法，分组长度和密钥长度都为 128 位，算法保密强度及性能与 AES 相当，该算法不公开	加密和签名算法
SM2	商密 2 号算法	国密标准的椭圆曲线加密算法，公钥密码算法，但比 RSA 更安全、更先进，在国家商密体系中用来替换 RSA 算法	加密和签名算法
SM3	商密 3 号算法	一种密码散列函数标准，在国家商密体系中主要用于数字签名及验证、消息认证码生成及验证、随机数生成等，其算法公开，其安全性及效率与 SHA256 相当	散列算法
SM4	商密 4 号算法	一种分组密码算法，在国家商密体系中主要用于数据加密，其算法公开，分组长度与密钥长度均为 128 位，加密算法与密钥扩展算法都采用 32 轮非线性迭代结构	加密和签名算法
SM7	商密 7 号算法	一种分组密码算法，分组长度为 128 位，密钥长度为 128 位，适用于非接触式 IC 卡	加密和签名算法
SM9	商密 9 号算法	一种标识密码算法，将用户的标识（如邮件地址、手机号码、QQ 号码等）作为公钥，省略了交换数字证书和公钥的过程，使得安全系统变得易于部署和管理，非常适合端对端离线安全通信、云端数据加密、基于属性加密、基于策略加密的各种场合	加密和签名算法

从上文得知，TPM 配置了非对称密码、散列函数、随机数产生器，更重要的是也配置了对应的硬件加速部件：

➢ **非对称密码**：TPM 一般采用 RSA 算法，密钥长度为 1 024/2 048 位，主要用于加密和数字签名，例如录入背书密钥（EK）、身份证明密钥（AIK）、存储根密钥（SRK）等。由于

非对称密码的产生比较耗时,因此需要采用硬件加速机制。

➢ **散列函数**:TPM 一般采用 SHA1 和 HMAC 算法,用于形成数据摘要以辅助数字签名和认证。

➢ **随机数产生器**:用于产生密码学随机数、对称密码的密钥以及认证过程中使用的随机数。

3. TPM 中的密钥管理

TPM 中的密钥分为背书密钥(Endorsement Key,EK)、身份证明密钥(Attestation Identity Key,AIK)、存储密钥(Storage Key,SK)、存储根密钥(Storage Root Key,SRK)、签名密钥(Signing Key,SIGK)、绑定密钥(Binding Key,BK)、继承密钥(Legacy Key,LK)和访问密钥(Access Key,AK)等。

➢ **背书密钥**:背书密钥是一种解密密钥,也是 TPM 的身份标志,因此是 RTR 的主要组成部分。EK 是一个 2 048 位的 RSA 密钥对。
- EK 在确定平台所有者时解密所有者的授权数据,还包括解密与生成 AIK 相关的数据。
- EK 只能由 TPM 制造商、平台拥有者、平台制造商而不是其他实体产生,每个 TPM 的 EK 都不同。
- EK 私钥以明文形式存储于 TPM 内部,公钥以证书形式管理。
- EK 不能用于数据加密和签名。
- EK 用于生成 AIK 和建立 TPM 平台的所有者。
- 应该由 TPM 的所有者来生成 SRK,使用 SRK 来加密和存储其他的密钥。

➢ **身份证明密钥**:是背书密钥的替代者,仅用于对 TPM 内表示平台可信状态的数据和信息(例如 PCR、时间戳、计数器、密钥可迁移数据等)进行签名和验证签名。
- 凡是经过 AIK 签名的实体,就表明它已经经过 TPM 的处理。
- 不能用 AIK 签名其他非 TPM 状态的数据,AIK 也不能用于加密。
- AIK 只能由 TPM 所有者在 EK 的控制下在 TPM 内部产生。
- 每个用户可以有多个 AIK,从而保证了平台隐私性。

➢ **存储密钥**:是 RSA 密钥对,用于对其他密钥(包括存储密钥自己)进行存储和加密保护。
- 根据密钥分级机制,低级密钥受高级 SK 加密保护,构成一个密钥树。
- SRK 处于密钥树顶端,产生于 TPM 内部,私钥也以明文方式存在于 TPM 内部。
- SK 也在 TPM 内部产生,使用时装入 TPM,但要经激活才能使用。

➢ **存储根密钥**:是存储密钥的特例,也是整个系统权限最高的存储密钥。
- SRK 在每个用户创建的时候生成,管理该用户的所有数据,也就是存储可信根(RTS)。
- 一个 TPM 只有一个 SRK。

- 所有其他的密钥都在 SRK 的保护之下(密钥树)。

➢ **签名密钥**:是非对称密钥,用于数据签名而不能用于加密。
- 遵循 RSA 签名密钥的标准,有若干种不同的长度。
- SIGK 在 TPM 内部产生,存储在 TPM 外部,使用时装入加解密模块。

➢ **绑定密钥**:用于加密小规模数据,在另一个 TPM 平台中解密。
- 由于 BK 使用平台所特有的密钥加密,所以与该平台绑定。
- BK 在 TPM 内部产生,存储在 TPM 外部,使用时装入加解密模块。
- BK 的用法与传统的非对称密钥加密相同。

➢ **继承密钥**:用在一些需要在平台之间传递数据的场合。
- 设置 LK 可以使 TPM 密码应用更加灵活。
- LK 在 TPM 内部产生,存储在 TPM 外部,使用时装入加解密模块。
- LK 在被用来签名或加密之后才能载入 TPM。

➢ **访问密钥**:是 TPM 中的对称密钥,用于加密保护 TPM 的会话,一般是一次一密。

上述密钥中,EK、AIK 和 SRK 必须是不可迁移密钥,其他的可以是可迁移密钥,也可以是不可迁移密钥。不可迁移密钥永久地与某个平台关联,也能够用来加密保护可迁移密钥,反过来却不行。不可迁移密钥由 TPM 内部产生,在产生之后就被打上了 TPM 的标记,安全性也非常高。同时不可迁移密钥可以被 TPM 签名,从而可以向挑战者或者用户证明其不可迁移的属性,进而证明其安全性。所谓可迁移就是密钥可以从一个可信计算平台转移到另一个可信计算平台。

TPM 中的密钥一般用在以下几个方面:

➢ **授权数据**:可以控制建立 TPM 所有权、密钥使用、对象迁移等行为。密钥的使用者必须拥有该密钥的授权数据验证码,只有通过验证才能使用。授权数据在密钥产生时设定,EK 与 SRK 的授权数据存储在 TPM 内部,其他的则随密钥一起存储。

➢ **平台关联**:将某个密钥与一个秘密随机数关联后,该密钥只能在该 TPM 内部使用,以此约束密钥的使用范围。TCG 指定了一部分 PCR 的值与产生的密钥的关联关系,使用密钥时 TPM 会核查指定的 PCR 的值。

➢ **绑定**:绑定是 TPM 用非对称密钥加密小规模数据的最基本方式。绑定指令先创建一个数据结构,将欲绑定的数据拷贝进该数据结构,最后用 BK 绑定加密该数据。

➢ **密封**:密封是使被加密的小规模数据与反映平台可信状态的各 PCR 的值关联起来的加密方法。只有 SK 才能用于密封数据。

表 15－2　TPM1.2 中定义的 PCR 寄存器

寄存器	存储内容	寄存器	存储内容
PCR0	BIOS 代码	PCR6	状态迁移
PCR1	硬件配置信息	PCR7	厂商使用

续表 15－2

寄存器	存储内容	寄存器	存储内容
PCR2	ROM BIOS 代码	PCR8～PCR15	存放供静态度量 OS 使用的数据
PCR3	ROM 配置信息	PCR16	存放供调试使用的数据
PCR4	IPL 代码	PCR17～PCR22	存放供动态度量 OS 使用的数据
PCR5	IPL 配置信息	PCR23	存放供应用程序使用的数据

此外，TPM 中还定义了 5 种证书，如图 15－14 所示。

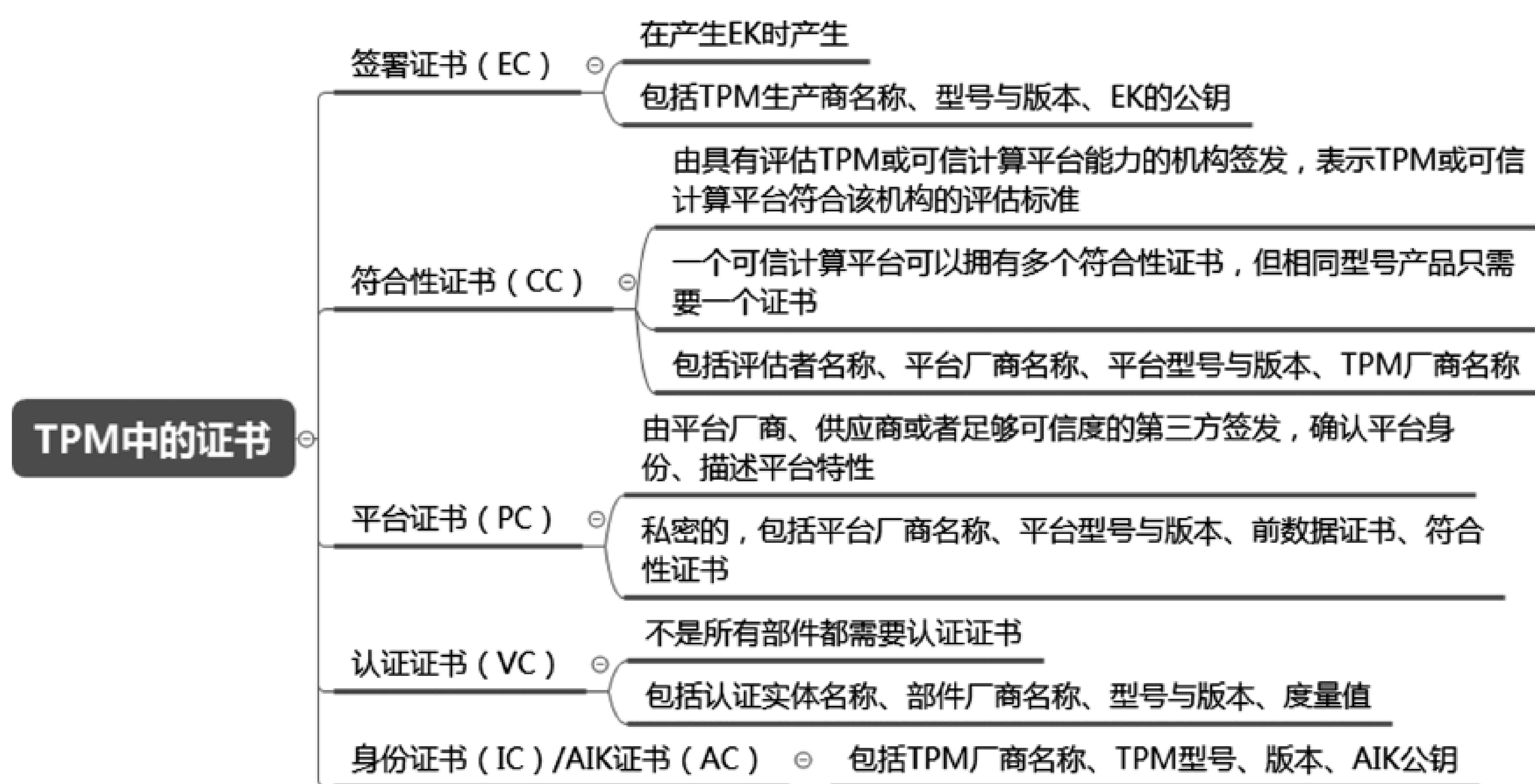


图 15－14　TPM 中的证书

访问 TPM 的对象需要得到授权，也就是说使用密钥和密封存储的数据时必须经过授权。代表授权的秘密信息是一个 160 位的 SHA1 算法的摘要值，即授权数据。其中 TPM 所有者和 SRK 的授权数据存储在 TPM 内部，其他实体的授权数据则随实体保存。

授权数据的建立和更改是通过下列协议实现的：

- **OIAP**：对象无关授权协议。
- **OSAP**：对象相关授权协议，与 OIAP 联合使用，用于将授权数据从请求者传递给 TPM，并建立授权会话上下文对象。OSAP 也是基本的授权数据验证协议。
- **DSAP**：委托相关授权协议。
- **ADIP**：授权数据插入协议。
- **ADCP**：授权数据修改协议。
- **AACP**：非对称授权更改协议。

15.4　可信软件栈

与 TPM 不同，可信软件栈（TSS）是一套软件系统，强调的是“栈”的层次关系。这个软件栈是为上层的可信计算应用提供访问 TPM 接口的软件系统，其体系结构如图 15－15 所示。

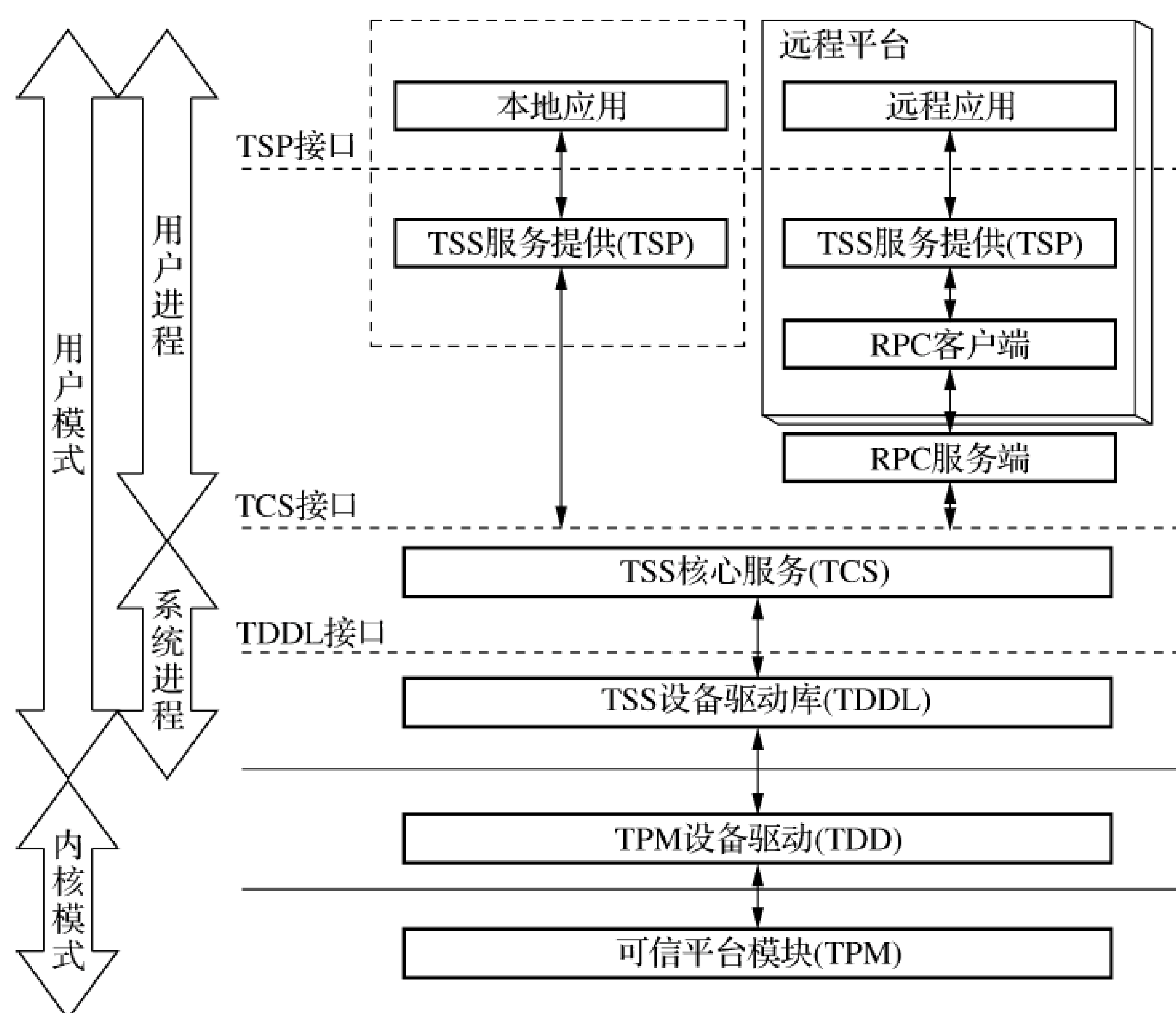


图 15－15　可信软件栈体系结构

可信软件栈从上到下大致可以分为 TSS 服务提供(TSP)、TSS 核心服务(TCS)和 TSS 设备驱动库(TDDL)三层：

➢ **TSS 服务提供(TSS Service Provider,TSP)层**
- 本地应用和 TSP 层之间是 TSP 接口(TSPI)层,TSPI 采用了面向对象的设计思想,支持通过句柄来找到对象实例。
- 本地应用使用这些对象实例,它们也被称为工作对象,可分为非授权对象(DAA 对象、散列对象等)和授权对象(TPM 对象、密钥对象、加密数据对象等)两类。
- 除了工作对象,TSP 还提供了上下文对象和策略对象。

➢ **TSS 核心服务(TSS Core Service,TCS)层**,TCS 以系统服务的形态存在于 TSS 中,其包括了以下功能：
- 管理 TPM 资源(如授权会话、密钥上下文交换等)。
- 提供 TPM 命令数据块产生器,将 API 转换成 TPM 可以识别的二进制代码。
- 提供一个全局密钥存储设备。
- 管理 PCR 事件。
- 同步来自于 TSP 层的访问请求。

➢ **TSS 设备驱动库(TSS Device Driver Library,TDDL)**:TDD 是个内核态组件,即 TPM 设备驱动,这是由 TPM 设备厂商提供的。用户态进程无法直接访问 TDD,因此设备厂商也同时提供 TDDL(可以与 TPM 设备驱动交互的 API 库)供应用程序调用,由 TDDL 与 TDD 打交道。

- TCS 往往是 TDDL 的唯一使用者。
- TDDL 是 TPM 的通信库和不同 TPM 的适配器。
- TDDL 为应用程序提供了不依赖于操作系统的接口。
- TDDL 希望由 TCS 来完成 TPM 命令的序列化。

根据上述分类,TSS 功能模块也可以分为 TSP 功能、TCS 功能和 TDDL 功能三部分:

➢ **TSP 功能**:包括上下文管理、策略管理、TPM 管理、密钥管理、PCR 管理、数据加解密管理、散列管理以及一些通用模块。
  - **上下文管理**:通过管理内部数据对象来协调各个模块的资源使用,包含了与对象执行环境相关的信息,这些都是与 TCS 交互时使用的。
  - **策略管理**:可以为不同的应用程序配置对应的安全策略,也可以为应用程序提供特定授权的秘密信息。
  - **PCR 管理**:PCR 用于建立系统平台的信任级别,API 通过 PCR 对象来使用 PCR 信息。

➢ **TCS 功能**:包括上下文管理、密钥证书管理、事件管理和参数块产生等。

➢ **TDDL 功能**:仅仅是用于管理 TDDL 接口。

15.5 基于可信计算的安全启动技术

可信计算技术在计算机安全启动、数字版权保护、软件防注入、身份盗用保护等诸多领域有着广泛应用。下面简单考察一下基于可信计算的计算机安全启动技术。

以 BIOS 方式启动的计算机为例,计算机启动时首先会执行作为可信度量根的一小段代码,这段代码可以是 BIOS 启动块甚至是整块 BIOS。利用这一小段代码作为校验启动过程中第一个模块的度量根,再以此为基础校验第二个模块、第三个模块。这种技术被称为**基于可信计算的安全启动技术**。

BIOS 一般包括 BIOS 启动块和 POST BIOS 代码两部分内容,且两者都可以单独更新,其中 POST BIOS 代码还可以分为初始部分和剩余部分两小段代码。同时,由于在 BIOS 启动块执行时主存尚未准备好,因此这个阶段只能使用主 CPU 的寄存器作为存储空间。如图 15-16 所示就是系统启动时基于可信密码模块的度量过程。

如图 15-16 所示,平台启动的同时了也启动了 BIOS 启动块和可信密码模块(TCM)。BIOS 启动块代码被散列生成摘要后调用 TCM 进行校验和存储,如果校验通过则以此为基础度量下一部分的代码,即 POST BIOS 代码的初始部分,度量的过程依然是散列生成摘要并要求 TCM 来进行校验并存储。初始部分通过验证后继续校验 POST BIOS 代码的剩余部分(此时主存已经初始化完成,因此这部分可以在主存中校验),最后验证 MBR 等模块。

可以看出,基于可信计算的安全启动技术是以可信密码模块为主要组件的一种层层校验式启动过程。



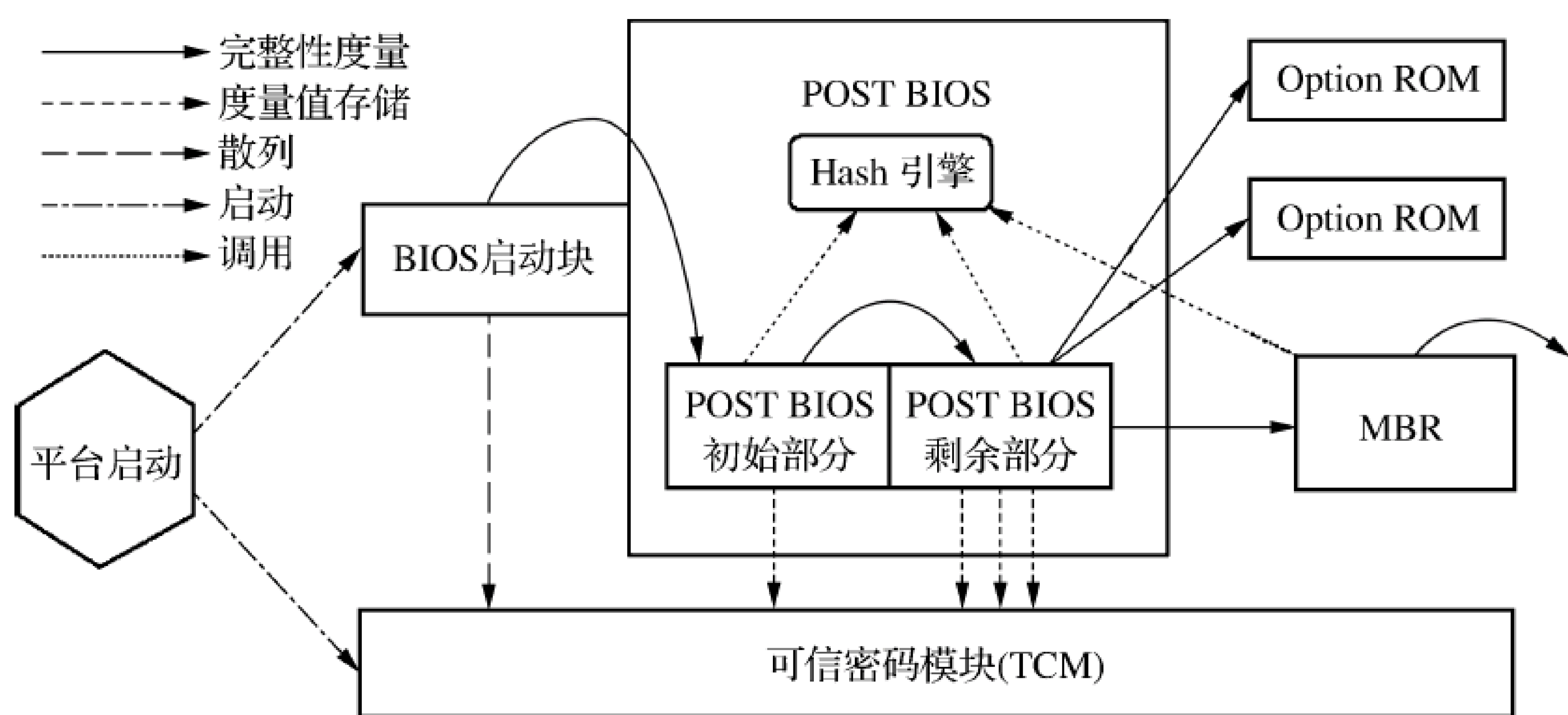


图 15－16　平台启动时的可信性度量

15.6　TrustZone 技术

消费类电子设备的安全保障一般通过以下两种技术方案实现：

- 外部挂接硬件安全模块，以保护自己的资源和密钥等数据不被任意访问，数据在外部安全模块和本地计算芯片上传输，保密程度不高。
- 计算芯片内部集成安全模块，数据传输通信放在了芯片内部，因此通信速度更快，保密程度更高。

基于第二种方案，ARM 提出了 TrustZone 技术，专门用于消费类电子设备的安全保密和可信计算。TrustZone 本质上是一种硬件架构和安全框架，其基本原理是将片上系统的软硬件资源划分为安全世界和非安全世界两个区域，类似于 X86 体系结构下的内核态与用户态，通过两个世界访问权限的差异保证资源的安全与可信。如图 15－17 所示，**非安全世界（Normal World）**和**安全世界（Secure World）**通过中间的监视模式（Monitor Mode）进行转换和通信。

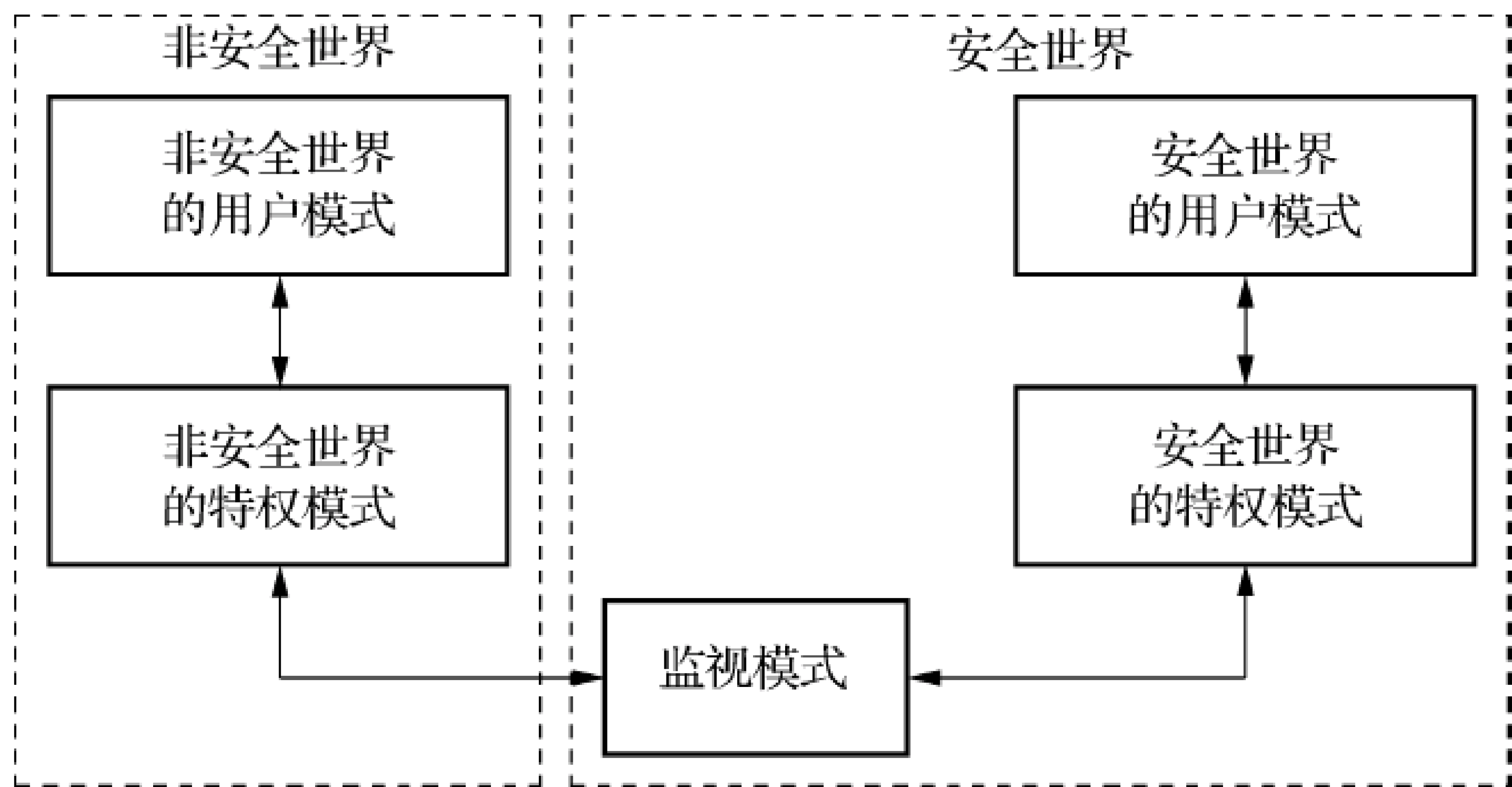


图 15－17　TrustZone 架构视图

在 TrustZone 架构中，将指纹识别、密码处理、加解密、安全认证等需要保密的操作运行

在安全世界,将操作系统、进程调度等不怎么需要保密的操作运行在非安全世界,二者通过中间的监视模式进行模式切换。

在 TrustZone 架构中一个 CPU 一般有两个虚拟核:安全核和普通核,核之间的通信是芯片内部的事情。安全核运行安全世界的代码,非安全核运行非安全世界的代码,两个虚拟的核以类似基于时间片线程调度的方式运行。这里要注意的是,安全核可以访问安全世界和非安全世界的资源,但是非安全核只能访问非安全世界的资源。

模式切换是通过 AMBA3 AXI 系统总线实现的,这种系统总线对于 TrustZone 的使用场景做了扩展改进,即在总线上针对每一个信道的读写增加了一个额外的控制信号位 NS(Non-Secure),总线上的所有主设备在发起新的操作时会设置这些信号,总线或从设备上的解析模块会对主设备发起的信号进行辨识,以确保主设备发起的操作在安全上没有违规。AMBA3 AXI 总线也为 TrustZone 架构提供了外设隔离的基础。

从非安全世界到监视模式是通过以下两种机制触发的:

➢ 软件执行安全监控指令(Secure Monitor Call);

➢ 硬件异常机制,如 IRQ、FIQ、External Data Abort、External Prefetch Abort 等。

从安全世界跳转到非安全世界必须经过监视模式的切换。因为如果不经过切换而直接跳转的话 CPU 的流水线和寄存器、缓存等可能还遗留了安全世界的数据,非安全模式就有可能从这些部件中直接获取秘密数据而造成安全隐患,给侧信道攻击带来便利,因此监视模式是二者切换的必由之路。

15.7 TXT 与 SGX

Intel 支持两种可信计算技术,即可信执行技术(Trusted Execution Technology,TXT)和软件保护扩展技术(Software Guard Extensions,SGX)。其中,TXT 出现的时间很早,并且是保护整个系统的技术,其范围包括 BIOS、CPU、BootLoader、OSLoader、OS 等组件,其信任链也很长;SGX 则主要是保护用户进程中的代码和数据的,并不关心 BIOS 或者 CPU 是否可信,也没有信任链的概念了。

1. Intel TXT 技术

Intel TXT 包含了一组新的安全指令,并且以 CPU 中的某段微码作为 CRTM,同时也支持 Intel 的虚拟化技术。其中支持 TXT 的芯片组拥有一组专门的 TXT 寄存器,并且被设计为一个增强型架构,同时对 TPM 的访问也是受控的(这里的 TPM 主要是用于提供存储空间和度量值管理)。

支持 TXT 的 BIOS 支持配置平台的安全模式,并提供了一个认证代码模块(ACM),这是 TXTBIOS 的核心。ACM 是芯片组厂商提供的,可以执行安全检查和注册,在处理器中的专用安全内存空间中以最高的权限执行。按照度量阶段的不同 ACM 可分为两种类型:

➢ BIOSACM:用于度量 BIOS 并执行多个基于 BIOS 的安全功能。

➢ SINITACM:用于执行操作系统的安全启动。

支持 TXT 的处理器启动时的度量序列如图 15－18 所示。

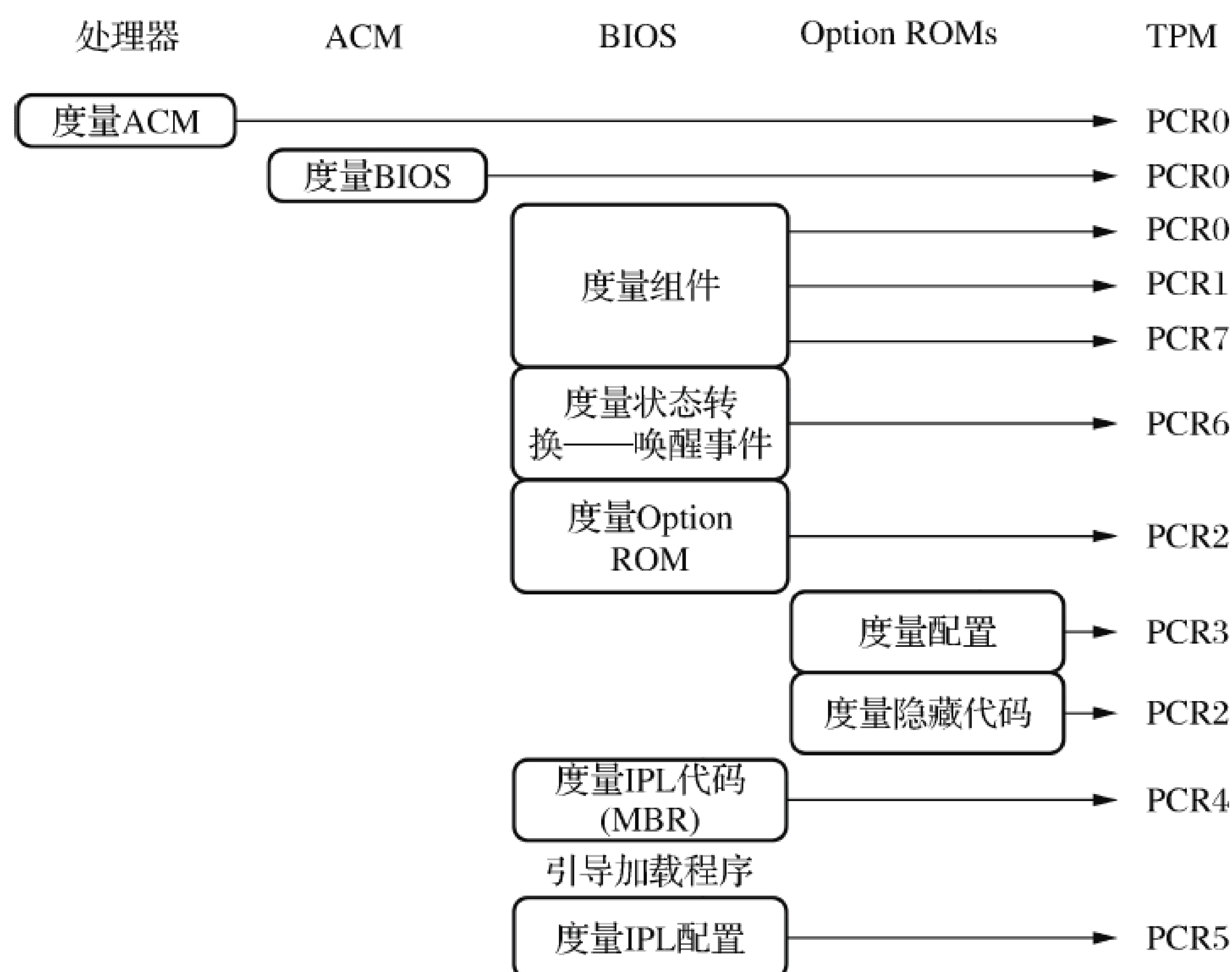


图 15－18 支持 TXT 的 CPU 的静态度量执行序列

2. Intel SGX 技术

Intel SGX 技术同样也包含了一组新的指令集,其 TCB 是基于硬件的,并支持严格的内存访问机制。其基本原理是在应用进程的虚拟地址空间中划分出一部分被保护的区域,为重要、敏感的代码和数据提供机密性和完整性保护。基于上述原理,要求在系统调用 SGX 指令之前必须支持内存分页机制并处于保护模式中。

在介绍 SGX 技术之前先来看看什么是 Enclave 和 EPC。

Enclave 可以理解为一个小黑盒,加载到这个小黑盒子中的数据和代码必须被度量,并且不允许外部软件访问小黑盒中的内容,因此可以看出 Enclave 是一个代码和数据运行的安全环境,可以杜绝恶意程序对应用进程代码和数据的访问。其特征可以概括如下:

➢ 具有自己的代码和数据;
➢ 提供机密性和完整性保护;
➢ 具有可以控制的入口点;
➢ 支持多线程;
➢ 对应用程序的内存空间具有最高的访问权限。

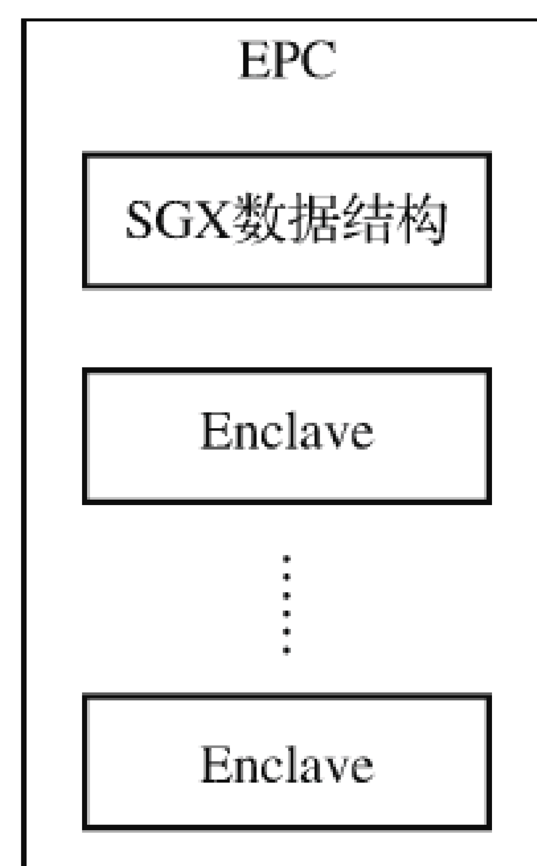


图 15－19 EPC 与 Enclave 的关系

EPC(Enclave Page Cache,Enclave 页面缓存)是 Enclave 驻留的内存区域,因此这是一块被保护的物理内存,专门存放 Enclave 和 SGX 的相关数据结构,如图 15－19 所示。并且 EPC 区域中的内容会被加密,只

有当这些内容进入 CPU package 时才会解密。

当 CPU 访问 Enclave 中的代码/数据时,CPU 会自动切换到一个新模式——Enclave 模式,该模式会强制对每一次的内存访问进行额外的检查,这种检查是由硬件实现的,因此其速度比较快。这种检查被称为 EPCM(Enclave Page Cache Map,Enclave 页面缓存映射)检查:检查请求访问的页面是否属于正在运行的那个 Enclave,也就是说确保只有 Enclave 中的代码才能访问该 Enclave 中的内容。

因此,SGX 技术的本质就是解决如何管理 Enclave 和如何访问 Enclave 中的数据/代码。可以分两个方面来回答上述问题:

(1) 如何解释内存访问的语义?

SGX 支持物理上锁住 EPC 内存区域,使外部进程无法访问。同时针对汇编指令的访问会执行以下检查:

➢ 检查处理器是否运行于 Enclave 模式。

➢ 检查访问地址是否在 Enclave 地址空间中,并检查物理地址是否在 EPC 内存区域中。

➢ 执行 EPCM 检查。

根据上述的检查结果,对于内存的访问可能产生如下的结果:

➢ 运行于非 Enclave 模式的 CPU 访问 EPC 之外的内存,这种情况下 CPU 按照保护模式执行访问。

➢ 运行于非 Enclave 模式的 CPU 访问 EPC 内部的内存,这种情况下将其视为引用了不存在的内存,等同于非法操作而拒绝。

➢ 运行于 Enclave 模式的 CPU 访问的内存页面不在当前 Enclave 虚拟地址空间,但处于 EPC 的内存区域范围,这种情况下依旧将其视为引用了不存在的内存。

➢ 运行于 Enclave 模式的 CPU 访问 EPC 以外的代码/数据,此时需要做进一步检查。

➢ 运行于 Enclave 模式的 CPU 访问当前 Enclave 内存区域,这是可以通过的。

(2) 如何实现内存地址映射?

EPC 中的内存依旧以 4KB 大小的页面为访问单位,页面的控制信息存放于 EPCM 中,这是一个硬件结构,并且只能由 PMH(Page Miss Handler,缺页处理器)硬件模块来访问,相当于在保护模式的基础上又增加了一层访问控制。其工作原理与内存页面表管理大致相同,每一个 EPCM 项代表一个 EPC 内存页面,并且所有的 EPCM 项构成了一个一维数组。

因此,综上所述,Enclave 是 SGX 技术中保护数据/代码机密性与完整性的关键。当应用程序申请一个小黑盒(Enclave)空间时自然要进行页面分配、数据拷贝等操作,因此申请 Enclave 的最后一步要对 Enclave 的完整性进行度量验证:是否有特权软件在创建过程中篡改了数据?是否有多与分配的内存页面?是否复制了恶意代码等。执行完上述验证后,SGX 会最终形成一个创建序列的度量结果集,并保存于 Enclave 的控制结构中。

SGX 再通过一条初始化指令将上述结果集与由 Enclave 的所有者签名过的证书中的完整性参考值作对比:

- 如果匹配，则会将证书中的所有者公钥进行散列计算，并作为密封身份保存于 Enclave 控制结构中；
- 如果不匹配，则返回失败结果。

本章小结

本章较为详细地介绍了可信计算相关的技术。

首先介绍可信计算的基础——加解密技术，以及可信计算的相关概念和信任链的原理，这是可信计算运行的基础。

然后介绍可信平台模块（TPM）的概念。TPM 是可信计算的核心模块，包括各种加解密引擎、存储器、数字证书等。

可信软件栈是可信计算的软件层次的实现，包括了向应用进程提供的服务、接口、硬件设备驱动库等重要组件。

接下来介绍了基于可信计算度量和验证机制的系统安全启动技术，该技术可以用于服务器、PC 以及各种物联网设备中。

最后分别介绍了 ARM 提出的 TrustZone 技术和 Intel 针对不同的保护对象而提出的 TXT、SGX 技术，这些技术都是芯片厂商针对具体处理器而提出的可信计算的保护方案。

第二区间

Windows驱动体系

第16章　Windows 设备驱动框架

Windows 设备驱动是内核中非常重要的模块,其地位相当于外围设备的应用使能软件。换句话说,没有设备驱动,就无法正常使用外围设备。Windows 设备驱动一般是 SYS 类型的文件,这类文件也是 Windows PE 文件体系下的一种,有自己的入口函数和加载地址。

驱动从广义上分为两种。一种是硬件驱动,这种驱动与具体设备绑定关联,负责翻译内核(例如 I/O 管理器)的 I/O 请求,并屏蔽不同硬件的差异性。例如显卡驱动负责将绘图数据解释成点、线、面并绘制到 HDMI 或 VGA 输出口上,但是 Intel 有 Intel 的显卡驱动,NVIDIA 有 NVIDIA 的显卡驱动,彼此之间不能通用;磁盘驱动负责将文件系统的 I/O 请求翻译成对磁盘的读写操作;而网卡驱动则负责网络请求包在网卡上的发送、接收和缓存等工作。

还有一种是纯软件驱动,这类驱动不与具体硬件设备绑定或关联,也不负责屏蔽不同厂家硬件设备的差异性。例如 TCP/IP 协议栈驱动(包括 tcpip. sys、TDI. sys 等)负责 OSI 参考模型中从第二层到第四层的协议解析与适配工作。

图 16 - 1 显示了 Windows 7 系统下的各驱动。

系统修复 | 进程管理 | 内核模块 | 内核相关 | 钩子 | 应用层 | 文件 | 注册表 | 启动项 | 系统服务 | 网络 | 硬件温度检测 | 关于和爱心捐助

驱动名	驱动类型	基地址	大小	驱动对象	驱动路径	文件厂商
tpm.sys	一般驱动	0xfffff88007d90000	0x27000	0xfffffa8007831e70	C:\Windows\system32\d...	Microsoft Cc
termdd.sys	一般驱动	0xfffff8800663e000	0x14000	0xfffffa80074d1e70	C:\Windows\system32\d...	Microsoft Cc
TeeDriverx64.sys	一般驱动	0xfffff88007b34000	0x32000	0xfffffa800784abf0	C:\Windows\system32\D...	Intel Corpor
tdx.sys	一般驱动	0xfffff880043d8000	0x22000	0xfffffa800749f980	C:\Windows\system32\D...	Microsoft Cc
TDI.SYS	一般驱动	0xfffff880043cb000	0xd000	-	C:\Windows\system32\D...	Microsoft Cc
tcpipreg.sys	一般驱动	0xfffff880037b7000	0x12000	0xfffffa8008123530	C:\Windows\system32\d...	Microsoft Cc
tcpip.sys	一般驱动	0xfffff88001603000	0x1fb000	0xfffffa8007388680	C:\Windows\system32\d...	Microsoft Cc
SynTP.sys	一般驱动	0xfffff8800822b000	0xe8000	0xfffffa8007896bb0	C:\Windows\system32\D...	Synaptics In
SynaSmi.sys	一般驱动	0xfffff88006610000	0xa000	0xfffffa80074cae70	C:\Windows\system32\D...	Windows (R
swenum.sys	一般驱动	0xfffff88007e1b000	0x2000	0xfffffa80077cb4b0	C:\Windows\system32\d...	Microsoft Cc
storport.sys	一般驱动	0xfffff88001165000	0x64000	-	C:\Windows\system32\d...	Microsoft Cc
srvnet.sys	过滤驱动	0xfffff88003786000	0x31000	0xfffffa8009ce5e70	C:\Windows\system32\D...	Microsoft Cc
srv2.sys	过滤驱动	0xfffff88003c30000	0x68000	0xfffffa8008158150	C:\Windows\system32\D...	Microsoft Cc
srv.sys	过滤驱动	0xfffff88003c98000	0x97000	0xfffffa80081fc680	C:\Windows\system32\D...	Microsoft Cc
spldr.sys	一般驱动	0xfffff88001912000	0x8000	0xfffffa8007413de0	C:\Windows\system32\D...	Microsoft Cc
smss.exe	过滤驱动	0x47ed0000	0x20000	-	C:\Windows\system32\s...	Microsoft Cc
Smb_driver_Intel...	一般驱动	0xfffff8800835e000	0x13000	0xfffffa80077df870	C:\Windows\system32\D...	Synaptics In
shlwapi.dll	过滤驱动	0xfffffffffd1e0000	0x71000	-	C:\Windows\system32\s...	Microsoft Cc
shell32.dll	过滤驱动	0xfffffffffe420000	0xd8a000	-	C:\Windows\system32\s...	Microsoft Cc
sgx_driver.sys	一般驱动	0xfffff880083f1000	0xf000	0xfffffa800778abd0	C:\Windows\system32\D...	Windows (R
setupapi.dll	过滤驱动	0xfffffffffde70000	0x1d7000	-	C:\Windows\system32\s...	Microsoft Cc
secnvmeF.sys	一般驱动	0xfffff8800191a000	0xb000	0xfffffa800740b940	C:\Windows\system32\d	Samsung El

驱动数量：227，可疑驱动：0

图 16 - 1　Windows 7 系统下的各驱动

驱动程序一般存在于内核态内存空间,这是因为驱动程序作为设备的代理要处理输入输出中断。既然是中断,其优先级和资源访问权限都是很高的,且更重要的是驱动程序往往要与 I/O 管理器等内核组件频繁交互,因此大部分驱动程序必须放在内核态内存空间。不过随着驱动技术的发展,目前也出现了用户态驱动程序,例如 WDF 模型中的 UWDF 驱动程

序就是专门的用户态驱动程序,数据平面开发套件(DPDK)作为网络协议栈驱动的旁路替代者也是存在于用户态空间的。

驱动程序的加载分为静态和动态两种方式,在操作系统引导(该阶段适用于老式驱动的加载)和初始化阶段(执行 IoInitSystem,适用于即插即用式驱动的加载)的加载是静态的,而通过系统调用 NtLoadDriver 加载的方式是动态的,且动态方式只能加载老式(Legacy)驱动,即使是支持即插即用的"较新"的驱动也是按照老式驱动的方式进行加载。这些老式驱动是 Windows 系统用于兼容过时的软件或硬件而保留的驱动程序。

当前 Windows 设备驱动按照架构可以分为 NT 式驱动、WDM 驱动和 WDF 驱动三种模型,按照功能则可以分为常规型驱动和过滤型驱动。常规型驱动就是我们常说的功能型驱动,这类驱动完成设备端既有的功能;过滤型驱动用于在驱动栈中过滤或者拦截 IRP,往往是作为栈中的一个节点堆叠在驱动栈中的。

本章将按照图 16-2 所示的提纲,根据设备驱动架构的划分方法分别介绍三种驱动模型:NT 式驱动模型、WDM 驱动模型和 WDF 驱动模型。

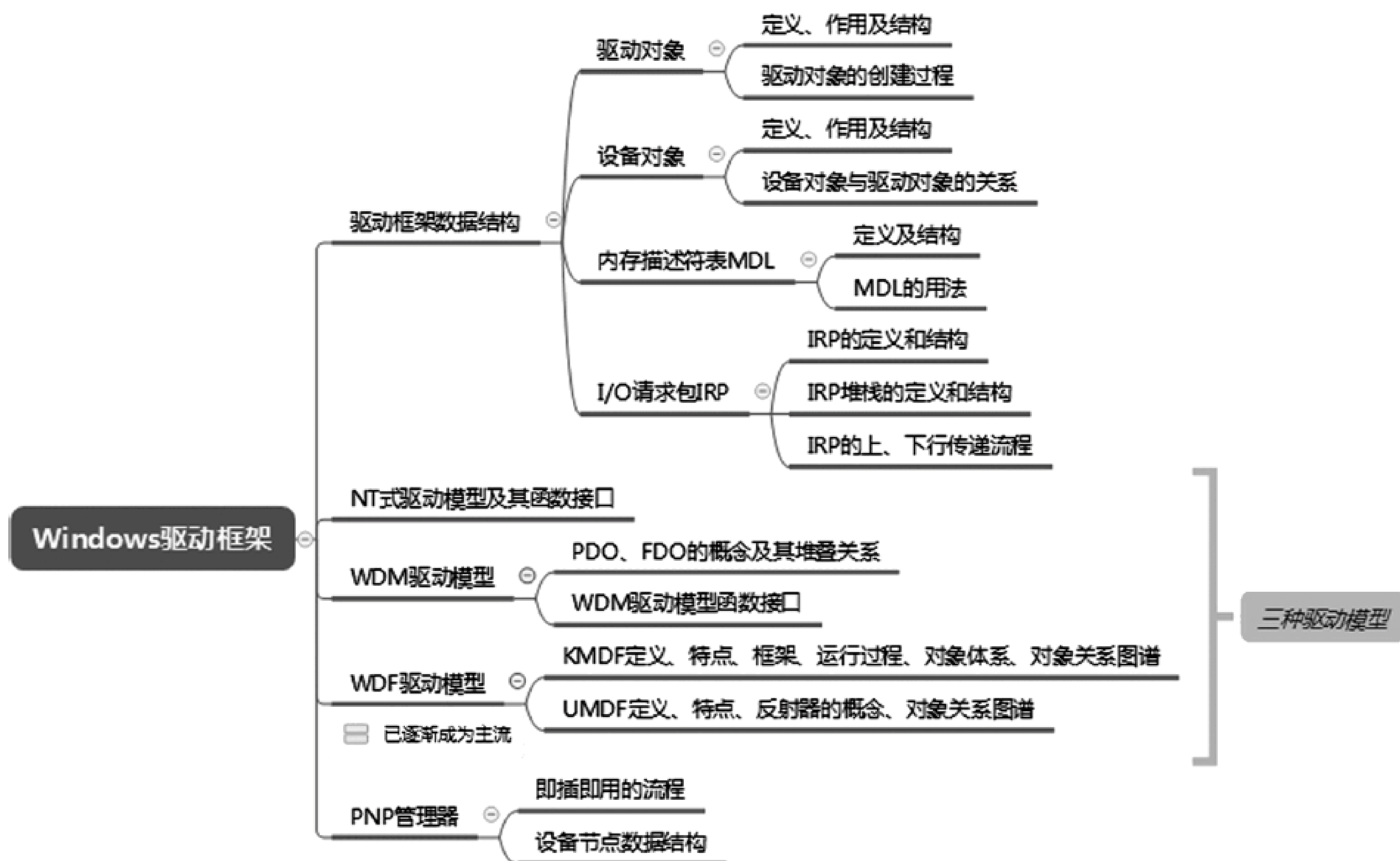


图 16-2　本章提纲

16.1　驱动框架的数据结构

16.1.1　驱动对象

每一个驱动程序都会以一个驱动对象实例(DRIVER_OBJECT)来表示,驱动对象是在驱动程序加载时由对象管理器创建的,包含了驱动对象的入口地址、驱动名称、映像大小、派遣

函数等关键信息。每个驱动对象会关联一个或多个设备对象(DEVICE_OBJECT),这是由驱动程序创建的,用于表示具体设备,例如磁盘、网卡等,但是一个设备对象只能属于一个驱动对象。驱动对象关联的多个设备对象形成设备对象链表,每个设备对象数据结构还包括一个扩展区域,这个区域存放了与特定设备相关的数据(DEVICE_OBJECT 只是存放了一些共性数据)。I/O 管理器向设备对象发送读写和控制请求,却是由驱动对象捕获和处理的。驱动对象还有个扩展结构体 DRIVER_EXTENSION,这个结构体存放了另外一些驱动程序的信息,但是比较有意义的是 AddDevice 函数指针,当操作系统发现一个新的设备实例时会自动调用该函数,因此 AddDevice 要完成设备实例相关的初始化工作。

从图 16-3 可以看出,有的驱动对象关联了一个设备对象,有的关联了多个。驱动对象 DRIVER_OBJECT 数据结构如下所示:

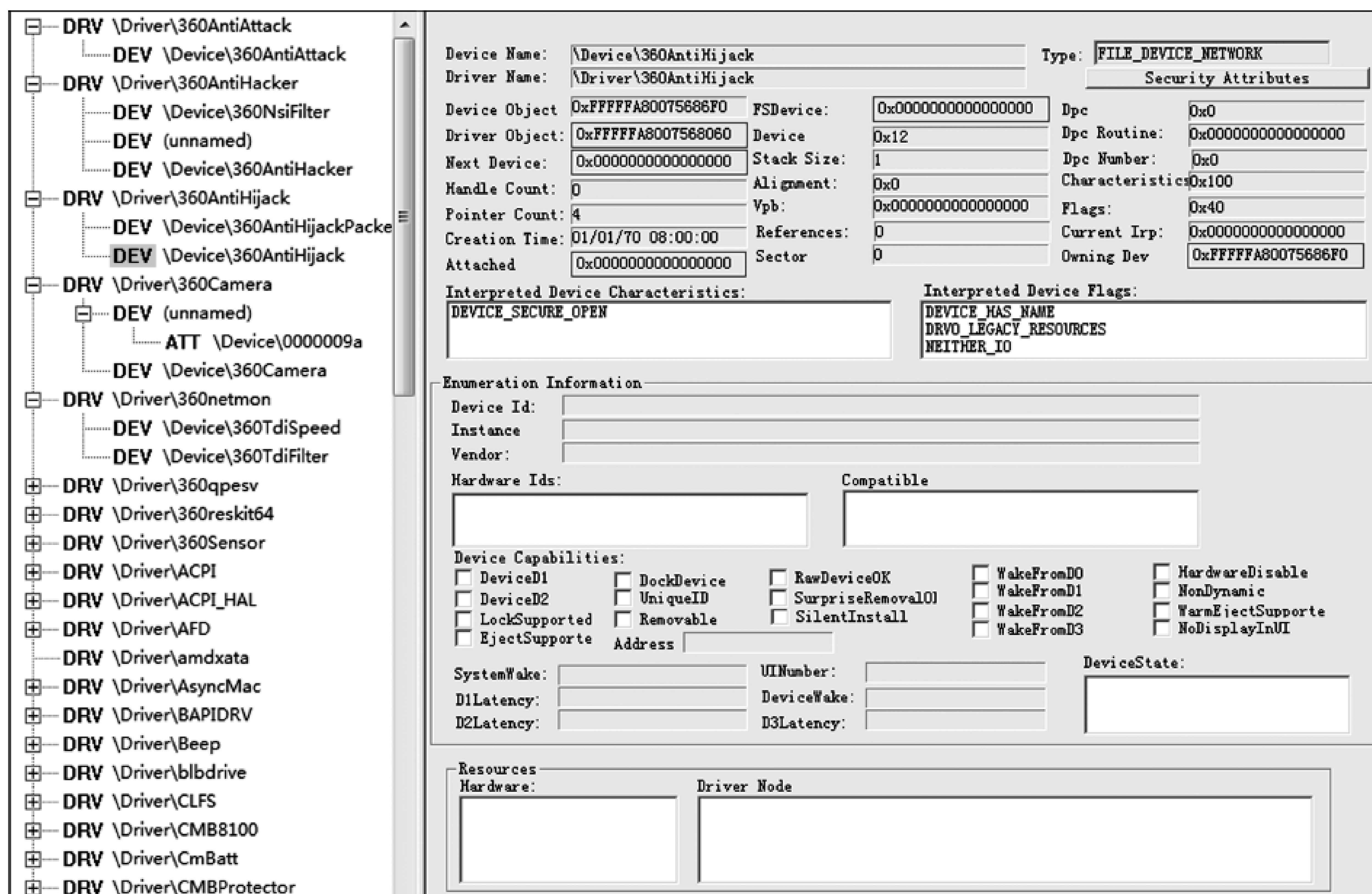


图 16-3 Windows 7 系统下驱动对象与设备对象的关联关系

```
typedef struct _DRIVER_OBJECT {
 CSHORT                Type;                //表明对象类型为驱动对象
 CSHORT                Size;                //驱动对象结构体大小
 PDEVICE_OBJECT        DeviceObject;        //驱动对象关联的设备对象或设备链表的第一个对象
 ULONG                 Flags;               //标志位
 PVOID                 DriverStart;         //驱动映像的基址
 ULONG                 DriverSize;          //驱动映像的大小
 PVOID                 DriverSection;       //Section 对象指针,指向驱动程序的可执行文件
 PDRIVER_EXTENSION     DriverExtension;     //驱动扩展区域指针
 UNICODE_STRING        DriverName;          //驱动程序路径和名称
 PUNICODE_STRING       HardwareDatabase;    //设备的注册表键名,决定了启动的顺序
 PFAST_IO_DISPATCH     FastIoDispatch;      //快速 I/O 函数指针
 PDRIVER_INITIALIZE    DriverInit;          //驱动加载时,I/O 管理器将 DriverInit 字段设置为驱
                                            //动的入口函数(DriverEntry)
 PDRIVER_STARTIO       DriverStartIo;       //指向 StartIO 函数指针,用于串行化操作
 PDRIVER_UNLOAD2       DriverUnload;        //驱动程序卸载函数指针
 PDRIVER_DISPATCH      MajorFunction[IRP_MJ_MAXIMUM_FUNCTION +1];    //派遣函数指针数组
```

```
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

从图 16－4 也可以看出，DRIVER_OBJECT 中也不是每个域都有意义，例如快速函数指针 FastIoDispatch 就可以为 0。

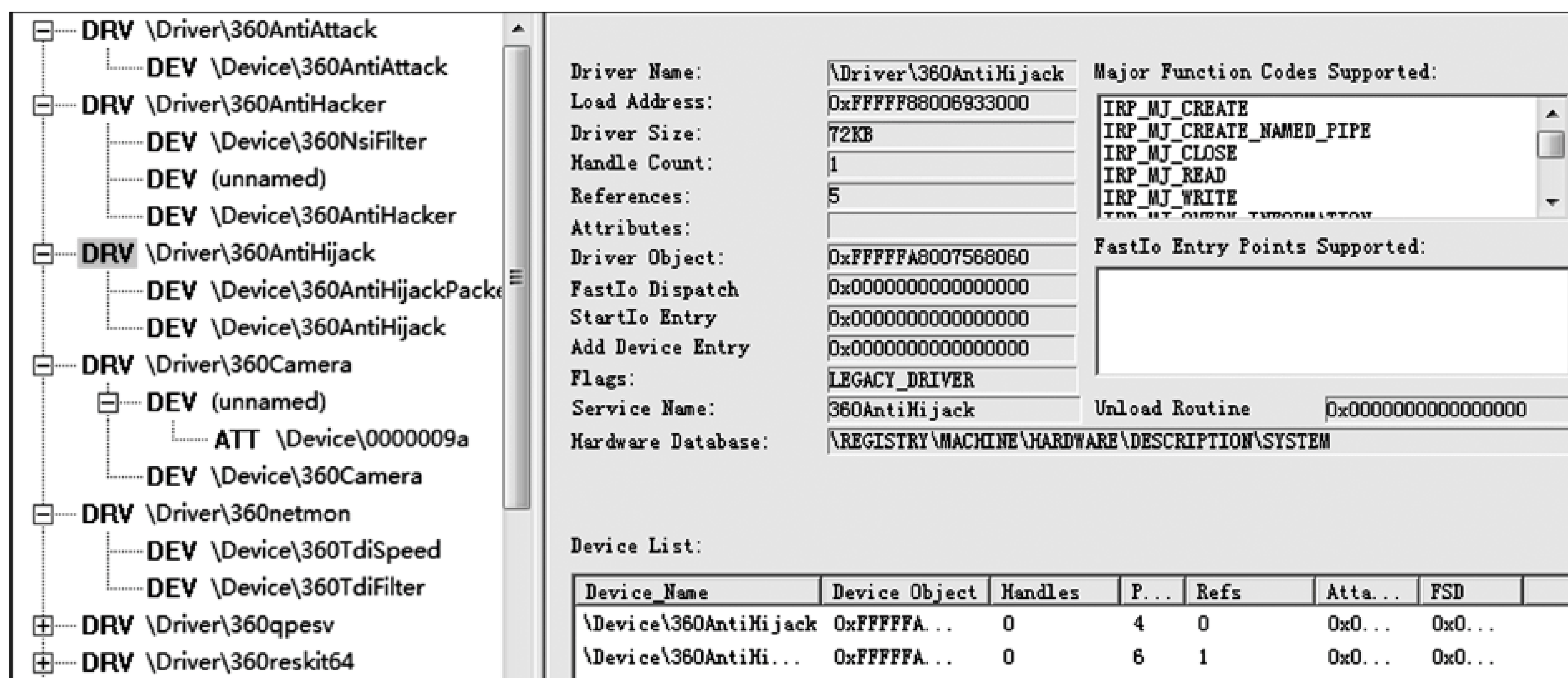


图 16－4　驱动对象数据结构视图

I/O 管理器调用 IopLoadDriver 函数加载驱动模块，驱动对象就是在这个函数中创建的，加载的具体步骤如下：

（1）根据驱动名称判断驱动是否已经被加载，已加载的驱动由系统中的驱动链表保存。

（2）如果驱动尚未被加载，则将驱动的可执行文件映射到内存中，当然虚拟内存管理器 VMM 也会检查该文件是否为合法的 PE 文件。

（3）I/O 管理器通过对象管理器创建该驱动的驱动对象并初始化（驱动对象存放于非分页内存池中）。

（4）I/O 管理器设置驱动对象的相关域值，包括驱动程序入口、驱动内存基址等。

（5）将该驱动对象挂入系统全局驱动列表中。

（6）I/O 管理器调用驱动的入口函数进行相关的初始化。

16.1.2　设备对象

设备对象 DEVICE_OBJECT 代表了设备实例，其数据结构如下所示：

```
typedef struct _DEVICE_OBJECT {
    CSHORT                    Type;               //表明设备对象
    USHORT                    Size;               //设备对象结构体大小
    LONG                      ReferenceCount;     //设备被打开的句柄数量，驱动被卸载时以此
                                                  //来判断是否还有设备被打开
    struct _DRIVER_OBJECT     *DriverObject;      //指向关联的驱动对象
    struct _DEVICE_OBJECT     *NextDevice;        //如果驱动对象关联了多个设备对象，该指针
                                                  //指向下一个设备对象
    struct _DEVICE_OBJECT     *AttachedDevice;    //堆叠挂载的设备对象，通常用于过滤型驱动
    struct _IRP               *CurrentIrp;        //当前正在处理的 IRP
    PIO_TIMER                 Timer;              //定时器指针
    ULONG                     Flags;              //标志位，组合按位或操作
    ULONG                     Characteristics;    //属性标志位，组合按位与操作
    __volatile PVPB           Vpb;                //与该设备对象相关的卷参数块指针（VPB），
```

```
                                                      //代表着一个已挂载的卷
    PVOID                  DeviceExtension;           //设备对象扩展区域指针,其大小和内容由驱
                                                      //动程序定义
    DEVICE_TYPE            DeviceType;                //设备种类描述
    CCHAR                  StackSize;                 //IRP 中的 stacklocation 的层数(最小值)
   union {
     LIST_ENTRY            ListEntry;                 //用于挂入链表
     WAIT_CONTEXT_BLOCK Wcb;
   } Queue;
    ULONG                  AlignmentRequirement;      //设备在大容量传输时需要内存对齐以加快传
                                                      //输速度,该域值表示使用内存时的对齐方式
    KDEVICE_QUEUE          DeviceQueue;               //等待处理的 IRP 的队列
    KDPC                   Dpc;                       //延迟过程调用函数指针
    ULONG                  ActiveThreadCount;         //未使用
    PSECURITY_DESCRIPTOR   SecurityDescriptor;        //指向安全描述表
    KEVENT                 DeviceLock;                //事件同步对象
    USHORT                 SectorSize;                //当设备是卷设备时表示卷中的分区字节数
    USHORT                 Spare1;                    //未使用
    struct _DEVOBJ_EXTENSION * DeviceObjectExtension; //设备对象扩展指针,用于存储设备状态信息
    PVOID                  Reserved;                  //未使用
 } DEVICE_OBJECT, * PDEVICE_OBJECT;
```

这里我们讲一下 DEVICE_OBJECT 结构中的 VPB 指针。一般来说,块存储设备都是与文件系统相关联的,一个块设备就是一个文件卷,而每个文件卷有可能使用不一样的文件系统(如 NTFS、FAT32 等),因此这类块设备对象需要通过一个数据结构与文件系统挂钩,表示这个块使用哪种文件系统进行解析,文件卷参数块 VPB 就是这样一个数据结构。

设备对象保存了设备的常规信息,而设备的特殊信息则保存于设备对象结构的扩展区域中(与驱动对象的扩展结构区分开)。这个扩展区域是由创建设备对象的驱动程序自己定义的,因此每种设备的扩展区域结构都不一样。由于这块区域是各个驱动程序通过 I/O 管理器创建的,并且位于内核中随时会被访问,也不允许缺页中断,因此保存在非分页内存池中。

驱动对象与设备对象的连接关系如图 16-5 所示,横向的"本层设备对象"即为设备链,纵向的通过 AttachedDevice 和 AttachedTo 指针串起来的即为设备栈,多用于驱动堆叠或驱动过滤场景。DEVICE_OBJECT 中的 AttachedDevice 域是个北向指针,指向更上层的设备对象。而设备对象扩展区域的 AttachedTo 域是个南向指针,指向更下层的设备对象。

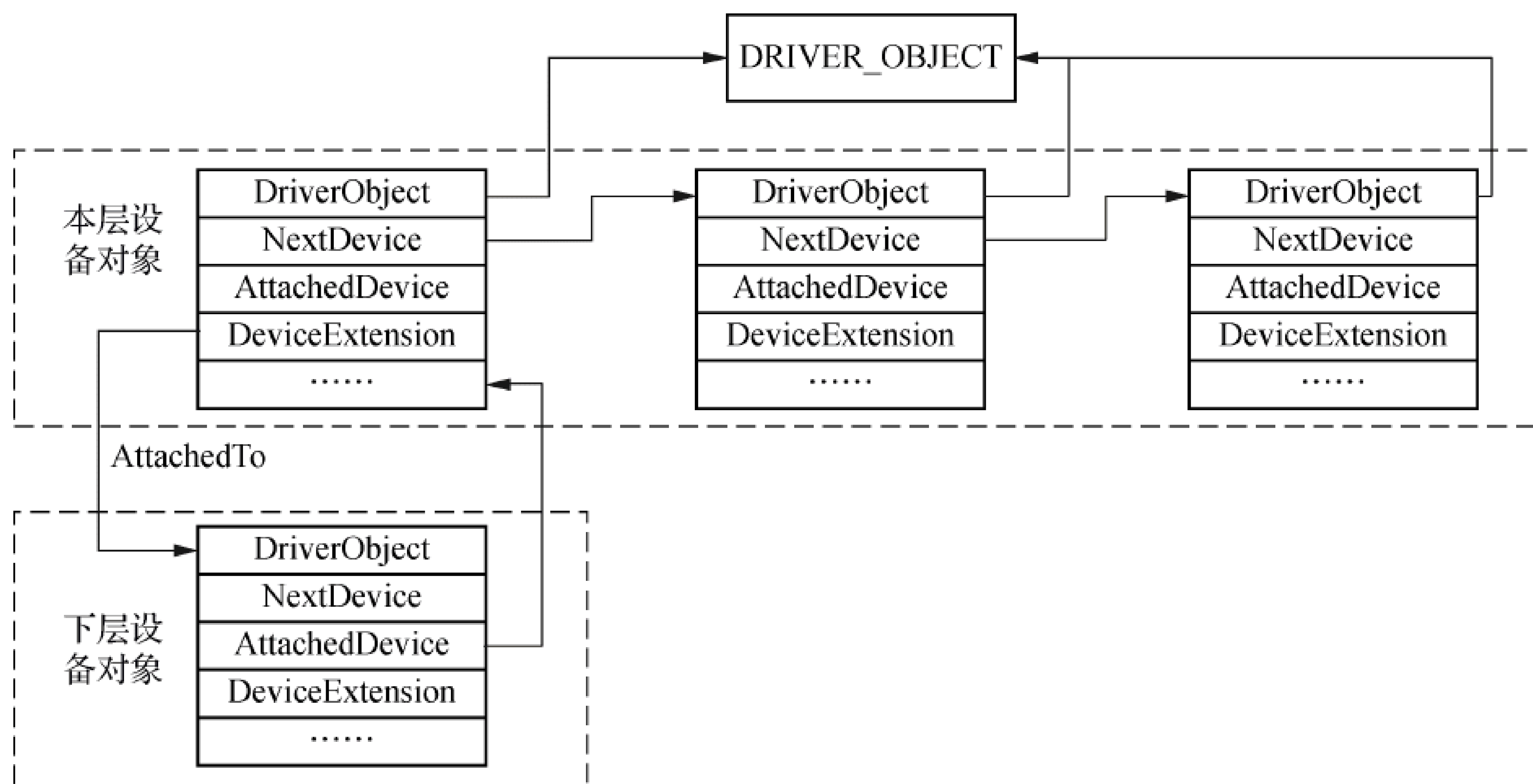


图 16-5　设备对象与驱动对象的连接关系

横向的设备链多用于串联同类型设备的多个实例，例如系统中有多块同类型网卡的场景，每个网卡就是一个设备实例，但却都共享相同的网卡驱动程序。

16.1.3　内存描述符表

内存描述符表（Memory Descriptor List，MDL）是一种特殊的数据结构，可以在物理页面中映射和描述在内核缓冲区虚存空间中的页面，多用于驱动程序与用户态进程之间的缓冲区数据读写，其本质就是"一块内存两份映射"，即用户态缓冲区的物理内存映射到内核态空间的虚拟地址中，如此一来一块物理内存就被映射到两个空间，对用户态空间的读写也相当于对内核态空间的读写，提高了 I/O 的效率。这种方法非常适合大数据块的交换，而针对小数据量的频繁读写，I/O 的开销就不可忽略不计了。

Windows 中 MDL 数据结构如下所示：

```
typedef struct_MDL {
  struct_MDL * Next;            //指向下一个 MDL 结构
  CSHORT Size;                  //该 MDL 的大小，从这个大小可以推算出 MDL 后面的数据结构的数组大小
  CSHORT MdlFlags;              //标志，保护属性、映射方式等
  struct_EPROCESS * Process;    //表示该 MDL 所属的进程
  PVOID MappedSystemVa;         //若映射到系统空间，则指明了 MDL 在系统空间映射的地址
  PVOID StartVa;                //映射的虚拟地址的开始页面的基址，该基址是内存页面对齐的
  ULONG ByteCount;              //该 MDL 描述的内存块的字节数
  ULONG ByteOffset;             //该 MDL 映射的虚拟地址的起始地址在 StartVa 页面中的偏移值
  } MDL, * PMDL;
```

在 MDL 数据结构的后面还紧跟着一个 PFN_NUMBER 结构数组，每个 PFN_NUMBER 结构占 4 个字节，代表了物理内存页面的页面号，也叫页帧号（Page Frame Number，PFN），PFN_NUMBER 结构数组中的页帧号用于描述 MDL 缓冲区中的一个物理页面。从 MDL.ByteCount 域可以推算出 MDL 描述的缓冲区覆盖了多少个物理页面。要注意的是 ByteCount 只是描述了缓冲区涵盖的字节数，占用的页面数则是 4 KB 对齐的，哪怕最后多余出一个字节，也要占用一个页面，如图 16－6 所示。

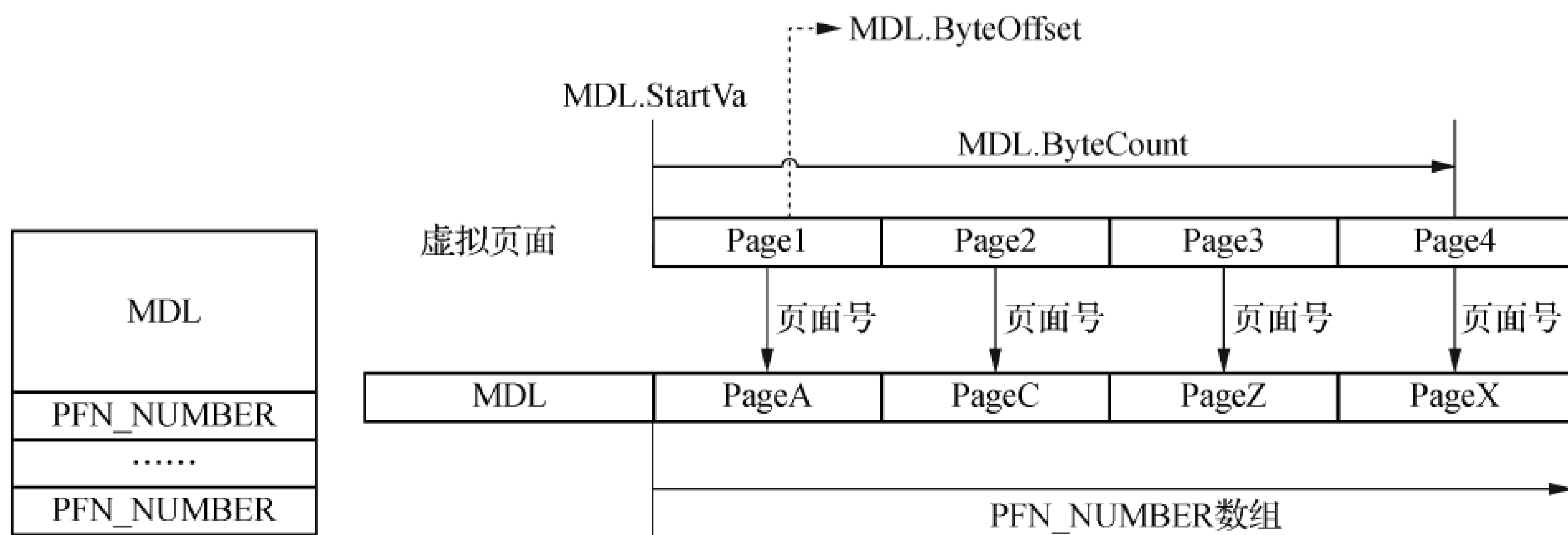


图 16－6　MDL 数据结构及其若干域的视图

总的来说，内核态与用户态进程之间的数据交换有三种方式：

➢ **缓冲方式（DO_BUFFERED_IO）**：创建一个中间缓冲区，用户态空间与内核态空间交换数据的时候，以该缓冲区为中转站，因此每次读写都会涉及两次内存拷贝和读写操

作。但是这种方式是进程上下文无关的,即缓冲区读写的双方没有必要是一个进程。这种方式比较适合少量数据块的交换。

- **直接方式(DO_DIRECT_IO)**:这也是 MDL 方式,创建 MDL 数据结构,即采用一块物理内存同时映射到用户态空间和内核态空间,因此只需要一次内存拷贝和读写操作。这种方式也是进程上下文无关的,比较适合大数据块的交换。
- **其他方式**:将用户态内存缓冲区的指针通过 IRP 传递到驱动程序中,本质上也是一块内存同时允许用户态空间和内核态空间读写,也只需要一次内存拷贝和读写操作,但这种方式要求二者在同一进程内,即进程上下文有关,这在大多数驱动程序中是很难做到的。

在 Windows 中,MDL 的创建函数为 IoAllocateMdl,创建完成后还要使用 MmInitializeMdl 方法对其进行初始化,而 MmBuildMdlForNonPagedPool 则负责为 MDL 后面的 PFN_NUMBER 数组赋值,MmGetSystemAddressForMdl 用于获取用户态空间缓冲区在系统空间的映射基址。

一个 MDL 可以描述一个页面内的缓存,也可以描述多个页面内的缓存,但这要求该缓冲区是连续的。如果存在若干个不连续的缓冲区,则要用多个 MDL 来描述。MDL 结构体的域中有 Next 指针,用于将 IRP 内的缓冲区映射描述符串联成一个链表。当然,物理页面号不一定是连续的,就像虚拟内存页面的连续页面并不一定也映射到连续页面号的物理页面中。

16.1.4 I/O 请求包

I/O 请求包(I/O Request Packet,IRP)是 I/O 管理器与驱动程序交互的信息载体。当上层的应用进程调用 Windows API 进行设备读写控制操作时,Windows API 调用系统服务函数将这些请求传递给操作系统的 I/O 管理器,I/O 管理器再把这些 I/O 请求翻译成 IRP 并投递给相应的驱动程序,最后由驱动程序调用对应功能码的派遣函数。当操作完成时,IRP 的完成请求沿着 IRP 的 StackLocation 顺序逆流而上,向上层层调用每层堆栈事先设置的完成函数,最终回到 I/O 管理器。

IRP 是对上层应用程序 I/O 功能的翻译,是 I/O 管理器和驱动程序中的“特殊语言”。其类型也对应了应用进程对于底层设备的操作要求的类型,而这些操作的 Windows API 就是形如 CreateFile、ReadFile、WriteFile 这样的函数。

IRP 的操作类型也是驱动对象派遣函数数组的下标。例如 ReadFile 函数通过 I/O 管理器会生成类型为 IRP_MJ_READ 的 IRP,当 IRP 传送到驱动程序中时,驱动程序在派遣函数数组 MajorFunction 中寻找下标为 IRP_MJ_READ 的函数,并将这个 IRP 作为参数传递到该函数中执行,而这个数组又是存在于驱动对象的扩展区域里的。操作类型在派遣函数中也叫主功能号,对应地还有个子功能号存放在 IRP 首部之后的 I/O 堆栈数据结构中,它表示派遣函数的细分选项,例如 IRP_MN_REMOVE_DEVICE 就是 IRP_MJ_PNP 的子功能号。我们可以观察两者的命名规范,如图 16-7 所示:

➢ **IRP_MJ_XXX** 表示主功能号(Major);
➢ **IRP_MN_XXX** 表示子功能号(Minor)。

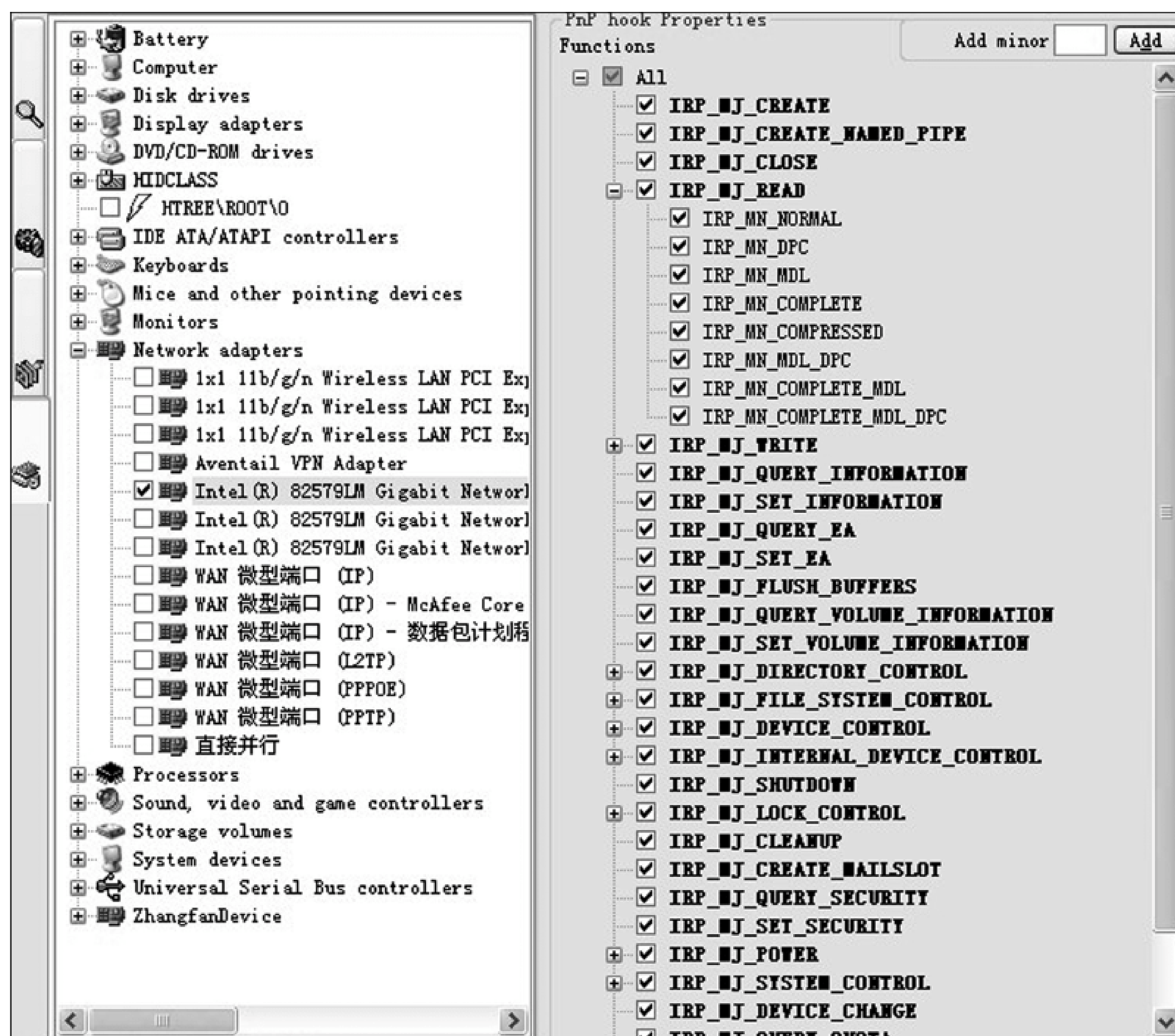


图 16－7　无线网卡驱动的主功能号和子功能号

这些功能号是由 I/O 管理器根据操作类型统一设置到 IRP 中的,通过 IRP 后面的 I/O 堆栈结构传递给驱动程序派遣函数。IRP 的类型可参阅表 16－1。

1. IRP 及其堆栈数据结构

IRP 数据结构如下所示:

```
typedef struct _IRP {
  PMDL                 MdlAddress;          //直接 I/O 方式时,该指针指向用户态空间的内存描述符表
  ULONG                Flags;               //对驱动程序只读的标志
  union {
    struct _IRP *      MasterIrp;           //关联式 IRP 的主 IRP
    PVOID              SystemBuffer;        //指向一个数据缓冲区,用户态缓冲区和内核态非分页内存池
                                            //中的数据交换
  } AssociatedIrp;
  IO_STATUS_BLOCK      IoStatus;            //IO_STATUS_BLOCK 结构体,驱动完成请求时设置该域
  KPROCESSOR_MODE      RequestorMode;       //指明该请求来源于用户态还是内核态
  BOOLEAN              PendingReturned;     //表明最低级的派遣函数返回了 STATUS_PENDING
  BOOLEAN              Cancel;              //表明 IoCancelIrp 函数是否已被调用
  KIRQL                CancelIrql;          //一个 IRQL 值,表明取消自旋锁是在该 IRQL 上获取的
  PDRIVER_CANCEL       CancelRoutine;       //取消函数的指针
 PVOID                 UserBuffer;          //对于 METHOD_NEITHER 方式的 IRP_MJ_DEVICE_CONTROL
                                            //请求,该域包含输出缓冲区的用户态虚拟地址

 union {
   struct {
     union {
       KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
```

```
            struct {
              PVOID    DriverContext[4];
            };
          };
          PETHREAD    Thread;
          LIST_ENTRY    ListEntry;
        } Overlay;
      } Tail;
    } IRP, *PIRP;
```

表 16－1　IRP 的类型

IRP 类型	类型描述	Windows API 发起者
IRP_MJ_CREATE	创建一个普通文件对象	CreateFile
IRP_MJ_CREATE_NAMED_PIPE	创建命名管道对象	CreateFile
IRP_MJ_CLEANUP	在关闭句柄时取消挂载的 IRP	CloseHandle
IRP_MJ_CLOSE	关闭文件对象句柄	CloseHandle
IRP_MJ_READ	从设备中读取数据	ReadFile
IRP_MJ_WRITE	向设备中写入数据	WriteFile
IRP_MJ_DEVICE_CONTROL	设备控制操作	DeviceIoControl
IRP_MJ_INTERNAL_DEVICE_CONTROL	控制操作（只能被内核调用）	无
IRP_MJ_QUERY_INFORMATION	获取文件的长度	GetFileSize
IRP_MJ_SET_INFORMATION	设置文件的长度	SetFileSize
IRP_MJ_QUERY_EA	获取文件的扩展信息	无
IRP_MJ_SET_EA	设置文件的扩展信息	CreateFile
IRP_MJ_FLUSH_BUFFERS	写输出缓冲区或者丢弃输入缓冲区	FlushFileBuffers/FlushConsoleInputBuffer/PurgeComm
IRP_MJ_QUERY_VOLUME_INFORMATION	查询磁盘卷信息	GetDiskFreeSpace/GetFileType
IRP_MJ_SET_VOLUME_INFORMATION	设置磁盘卷信息	SetVolumeLabel
IRP_MJ_DIRECTORY_CONTROL	查询某个目录下的文件和子目录信息	ZwQueryDirectoryFile
IRP_MJ_FILE_SYSTEM_CONTROL	文件系统控制操作	DeviceIoControl
IRP_MJ_SHUTDOWN	关闭系统	InitiateSystemShutdown
IRP_MJ_LOCK_CONTROL	锁控制请求	无
IRP_MJ_CREATE_MAILSLOT	创建邮件槽	CreateFile
IRP_MJ_QUERY_SECURITY	查询安全描述符	无
IRP_MJ_SET_SECURITY	设置安全描述符	无
IRP_MJ_POWER	电源管理操作	无

续表 16－1

IRP 类型	类型描述	Windows API 发起者
IRP_MJ_SYSTEM_CONTROL	为鼠标、键盘等提供 WMI 数据源	无
IRP_MJ_DEVICE_CHANGE	设备状态发生改变	无
IRP_MJ_QUERY_QUOTA	获取配额信息	GetQuotaState
IRP_MJ_SET_QUOTA	设置配额信息	SetQuotaState
IRP_MJ_PNP	即插即用消息，插拔操作发生时被触发，NT 式驱动不支持该消息类型	无

一个 IRP 是从非分页内存池分配的可变大小的数据结构，包括两部分：IRP 首部和 I/O 堆栈。其实上述 IRP 数据结构只是描述了 IRP 首部，包含 I/O 缓冲区指针、IRP 某些函数的指针等重要信息。紧邻着 IRP 首部的高址部分即 I/O 堆栈结构，这是一个 IO_STACK_LOCATION 结构的数组，数组大小由设备栈中的设备数量确定。每一个 IO_STACK_LOCATION 结构都保存着一个 I/O 请求的参数、功能码、I/O 请求对应的设备指针、I/O 完成函数指针（IoCompletion Routine）等信息，表示每一层堆栈结构都对应一个栈中的设备对象，每个设备对象都会对 IRP 进行处理，并且只允许设备对象访问本层的 IO_STACK_LOCATION，如图 16－8 所示。

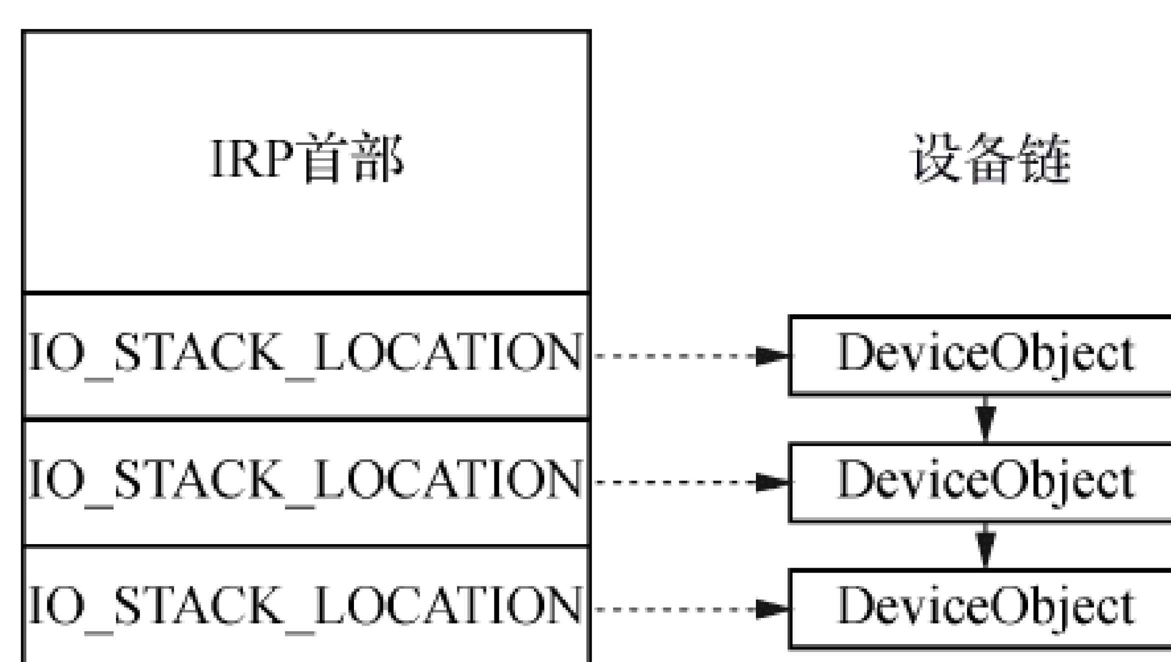


图 16－8　IO_STACK_LOCATION 与设备对象的对应关系

I/O 堆栈的数据结构 IO_STACK_LOCATION 如下所示：

```
typedef struct_IO_STACK_LOCATION {
  UCHAR                  MajorFunction;           //IRP 的主功能码
  UCHAR                  MinorFunction;           //IRP 的子功能码
  UCHAR                  Flags;
  UCHAR                  Control;
  union {
  ......
  } Parameters;                                   //多个结构体的联合,此处不详细描述
  PDEVICE_OBJECT         DeviceObject;            //与该堆栈单元关联的设备对象的地址
  PFILE_OBJECT           FileObject;              //IRP 目标文件对象的地址
  PIO_COMPLETION_ROUTINE CompletionRoutine;       //I/O 完成函数指针
  PVOID                  Context;
} IO_STACK_LOCATION, * PIO_STACK_LOCATION;
```

2. IRP 的下行传递流程

在一个 IRP 中,上层驱动负责为下层驱动设置堆栈位置指针。驱动程序可以为每个 IRP 调用 IoGetCurrentStackLocation 方法以获取指向其自身堆栈位置的指针,上层驱动程序需调用 IoGetNextIrpStackLocation 方法来获得指向下层驱动程序堆栈位置的指针。这些 IRP 相关的方法如表 16－2 所示,它们都是由 I/O 管理器提供的。

表 16－2　IRP 相关方法

IRP 方法	方法描述
IoStartPacket	将 IRP 发送给 StartIO 函数,插入设备 IRP 队列以串行化处理 IRP
IoCompleteRequest	IRP 处理完成
IoStartNextPacket	将下一个 IRP 发送给 StartIO 函数
IoCallDriver	将 IRP 传递给驱动程序
IoAllocateIrp	分配一个 IRP 数据结构
IoFreeIrp	释放一个 IRP 数据结构
IoGetCurrentIrpStackLocation	获取当前调用者的 I/O 堆栈指针
IoMarkIrpPending	标记 IRP 堆栈标志
IoGetNextIrpStackLocation	获取下一层驱动的 I/O 堆栈指针
IoSetNextIrpStackLocation	将 I/O 堆栈指针压入堆栈

上层驱动调用 IoCallDriver 方法将 IRP 向下传递,其参数 DeviceObject 域被设置为下层目标驱动的设备对象。当下层驱动完成 IRP 时,IRP 堆栈中的完成例程 CompletionRoutine(只能由上层驱动通过 IoSetCompletionRoutine 方法设置)被调用,I/O 管理器将上层驱动设备对象的指针传递给 CompletionRoutine。图 16－9 描述了从用户态到内核态整体的 I/O 请求走向。

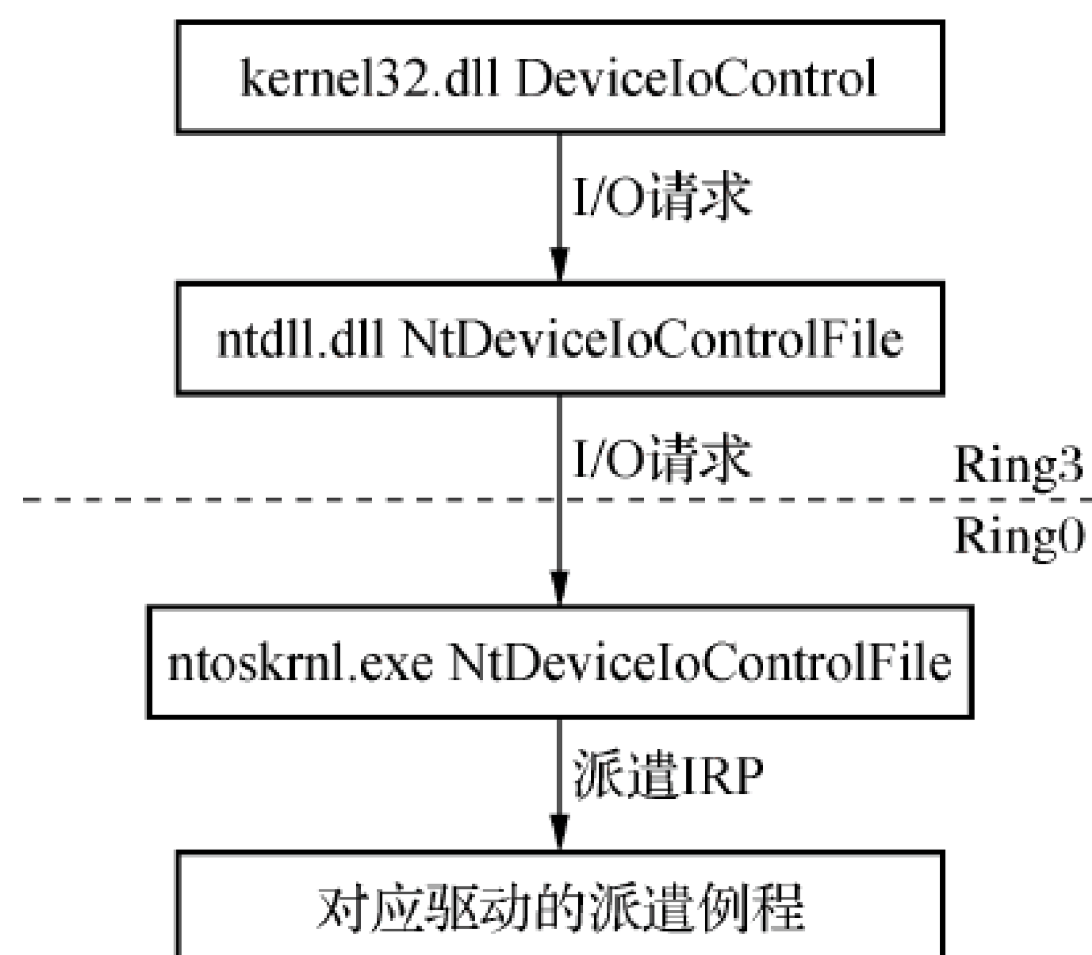


图 16－9　I/O 请求的流向

综上所述,IRP 的处理流程是这样的(以 ReadFile 函数为例):

（1）Windows API 将 I/O 请求透传给 I/O 管理器。

（2）I/O 管理器将 I/O 请求翻译成 IRP 并通过设备对象传递到驱动对象的派遣函数中。

（3）派遣函数将 IRP 按设备栈向下传递到最底层驱动，最底层驱动进行处理，读取网卡或磁盘中缓存的若干个数据包。

（4）驱动程序从 I/O 堆栈逆流向上返回到 I/O 管理器并产生 I/O 中断事件。

（5）I/O 管理器通过中断响应例程插入 DPC（ISR 的前半段安排了后半段的执行）。

（6）DPC 从驱动的缓冲区中读取网络或磁盘数据包。

（7）DPC 将这些数据包返回给应用进程。

从上述描述可以看出，IRP 的下行与上行两个方向上是严格遵守驱动堆栈顺序的，不能跳跃，如图 16－10 所示。

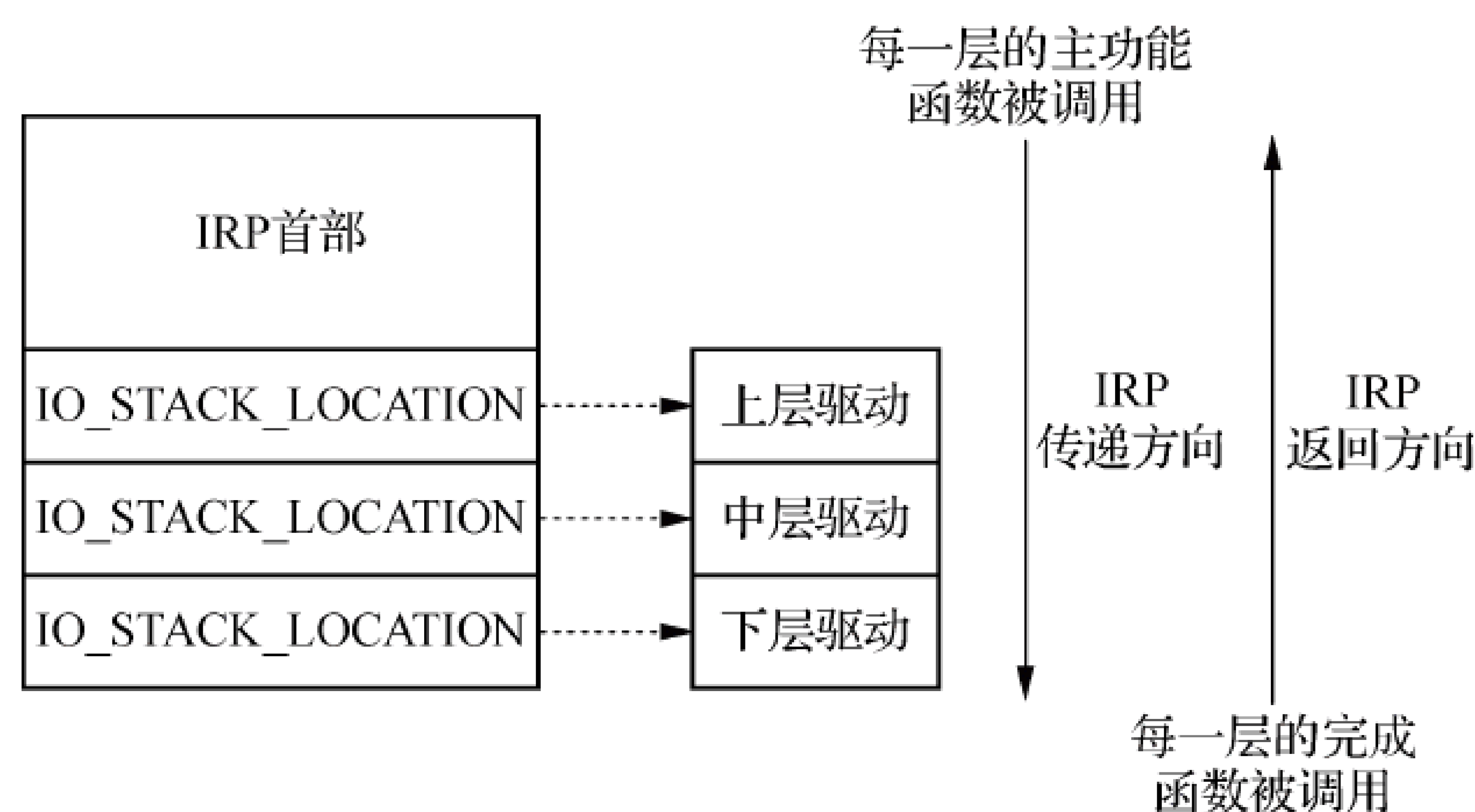


图 16－10　IRP 的传递和返回方向

3. IRP 的上行返回流程

我们在这里还要详细讲述一下 IRP 的完成与返回。

当 I/O 请求完成时，驱动程序要执行完成函数，将 I/O 的结果返回给 I/O 操作的发起者，这是完成函数 IoCompleteRequest 的使命。IoCompleteRequest 有以下两种被调用的时机：

- 如果是同步调用方式，I/O 完成时当前线程尚在主功能函数的执行过程中，因此 IoCompleteRequest 直接在主功能函数中被调用。
- 如果是异步调用方式，I/O 完成时主功能函数已经执行完毕，当前线程可能在其他上下文中，IoCompleteRequest 需要以 DPC 的方式被异步调用。

在 IRP 的 I/O 堆栈（IO_STACK_LOCATION 数据结构）中有个 CompleteRoutine 函数指针，这是本层驱动程序自我设置的善后函数执行的一个渠道。IoCompleteRequest 执行时，除了完成通知工作，也会从下向上逐层调用 CompleteRoutine 函数，以完成各层驱动程序自己设置的善后工作。

最后，IoCompleteRequest 通知 I/O 管理器本次 I/O 操作已结束，I/O 管理器则创建一个 I/O 完成的 APC 并投递到发起 I/O 请求的线程中，在线程切换的间歇这些 APC 被执行，I/O 的完成结果被回调到上层的应用进程中。这也是异步读写的一般返回步骤。

针对多层驱动堆叠的场景，每一层驱动都会对应一个设备对象，IRP在传递的过程中层层穿透每一层的I/O堆栈，本质上相当于逐层经过设备栈中的设备对象。I/O堆栈中存放着当前IRP的主副功能码和I/O完成函数的指针，请求发起时自上而下每层堆栈对应的功能函数被调用；请求完成时自下而上每层堆栈的I/O完成函数被调用。

IRP也支持取消操作，例如上层应用撤销了对设备的某次读写。IRP取消是由系统调用函数NtCancelIoFile完成的，该函数首先调用IoCancelIrp取消当前线程的IRP队列中的所有IRP，这些IRP是保存在KTHREAD. IrpList中的；之后要多次扫描这个队列以确保全部IRP能正常取消。由于IRP的取消有一定的滞后性，调用了IoCancelIrp并不一定能保证马上从队列中移除IRP，需要一定的执行时间，因此应该多执行几遍以确保取消操作的彻底性。

16.2 NT式驱动模型

NT式驱动是不支持即插即用(Plug-and-Play，PNP)功能的老式驱动程序，这类驱动可以通过INF配置文件加载，并且可以在系统中以系统服务的形式存在。

NT式驱动模型是Windows中最古老的驱动模型，WDM和后来的WDF模型都是基于NT式驱动模型发展而来的。NT式驱动包含以下接口函数：

➢ **驱动程序入口函数DriverEntry**：这是由驱动模块提供的入口函数，负责对驱动程序进行初始化操作，相当于DLL文件中的DllMain。该入口函数一般是在系统初始化的时候由系统进程(System)调用的，当然也可以通过系统调用NtLoadDriver方法动态载入后调用。在DriverEntry中既要创建驱动对象，也要设置卸载函数指针和IRP派遣函数指针数组。

➢ **驱动程序卸载函数DriverUnload**：这是由驱动模块提供的函数，负责卸载驱动、删除设备、删除符号链接等。所谓符号链接就是以绝对路径或者相对路径的形式指向其他文件或者目录，是文件/目录的引用，也是设备对象的别名，既可以被用户态应用进程识别，也可被内核态驱动程序识别。因此用户态进程与内核态驱动的通信在无法直接访问设备对象的情况下可通过符号链接实现。

同时，I/O管理器也为设备对象提供了若干接口：

➢ **设备对象创建函数IoCreateDevice**：这是由内核(I/O管理器)提供的函数，负责创建设备对象。设备对象的类型为IoDeviceObjectType(注意不是设备的类型，而是设备对象的类型)，这是供对象管理器使用的类型。

➢ **设备对象删除函数IoDeleteDevice**：这也是由I/O管理器提供的函数。

➢ **符号链接创建函数IoCreateSymbolicLink**：这是由I/O管理器提供的函数，用于将设备与符号链接进行绑定，如图16-11所示。

➢ **符号链接删除函数IoDeleteSymbolicLink**：这是由I/O管理器提供的函数，用于将设备与符号链接解除绑定。

Name	Type	SymLink
DR0	Device	
Partition0	SymbolicLink	\Device\Harddisk0\DR0
Partition1	SymbolicLink	\Device\HarddiskVolume1
Partition2	SymbolicLink	\Device\HarddiskVolume2
Partition3	SymbolicLink	\Device\HarddiskVolume3
Partition4	SymbolicLink	\Device\HarddiskVolume4

图 16－11　磁盘分区名与对应的符号链接

NT 式驱动模型的设备插入计算机系统后系统不会有任何提示，因而需要用户自己安装相应的驱动程序，也就是说这类设备不是“即插即用”的。

16.3　WDM 驱动模型

WDM（Windows Driver Model，Windows 驱动模型）也被称为 Windows 驱动程序模块，是从 Windows 2000 开始就被引入的一种驱动模型。WDM 源于 NT 式模型，除了支持即插即用功能外，还支持 WMI 和电源管理功能。因此，WDM 较 NT 式模型能够大大简化驱动程序本身的硬件检测、设备对象创建以及初始化等工作。

WDM 的驱动一般是分层的，对应的设备对象至少分为物理设备对象（Physical Device Object，PDO）和功能设备对象（Function Device Object，FDO）两部分，且 FDO 附加在 PDO 之上。

当某个外设被插入计算机的时候（例如鼠标、键盘等），PDO 会由总线驱动（总线驱动一般用于枚举设备，负责物理层的通信）自动创建。但是 PDO 不能独自使用设备，必须有上层功能软件系统的配合，FDO 就承担了功能软件的职责。插入后，系统提示检测到新设备，这时需要安装 WDM 驱动程序，如果微软已经提供了该种类设备的驱动，则操作系统会自行安装，否则会联网查找对应的驱动程序下载并安装。WDM 驱动程序除了要创建 FDO，还要将 FDO 附加到 PDO 之上，如图 16－12 所示。

在 FDO 与 PDO 之间或在 FDO 的上层可能还会有一个甚至多个过滤型驱动。这类驱动一般是由第三方程序安装的（例如杀毒软件或深度包检测软件等），其作用是截取流经 FDO 与 PDO 之间的数据包并进行过滤，要么增加包的内容，要么改变包的内容，要么监控包的内容，如图 16－13 所示。当然，不只是 WDM 可以安装过滤型驱动，NT 式驱动模型也可以安装，进一步说，只要具备设备对象并且设备对象可堆叠的驱动程序都可以附加过滤型驱动，因为堆叠本质上是设备的堆叠。

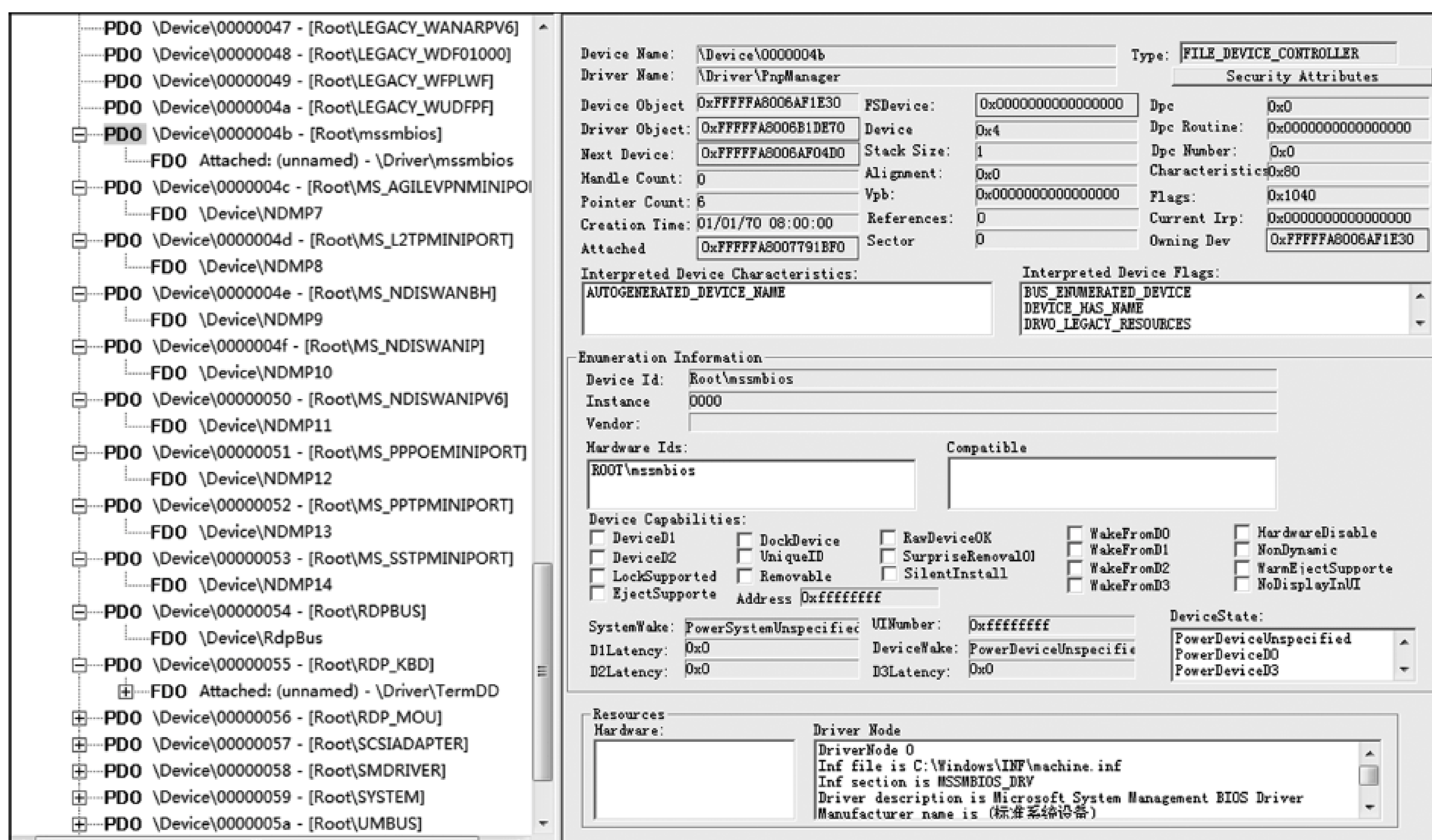


图 16－12　PDO 与 FDO 的附加关系

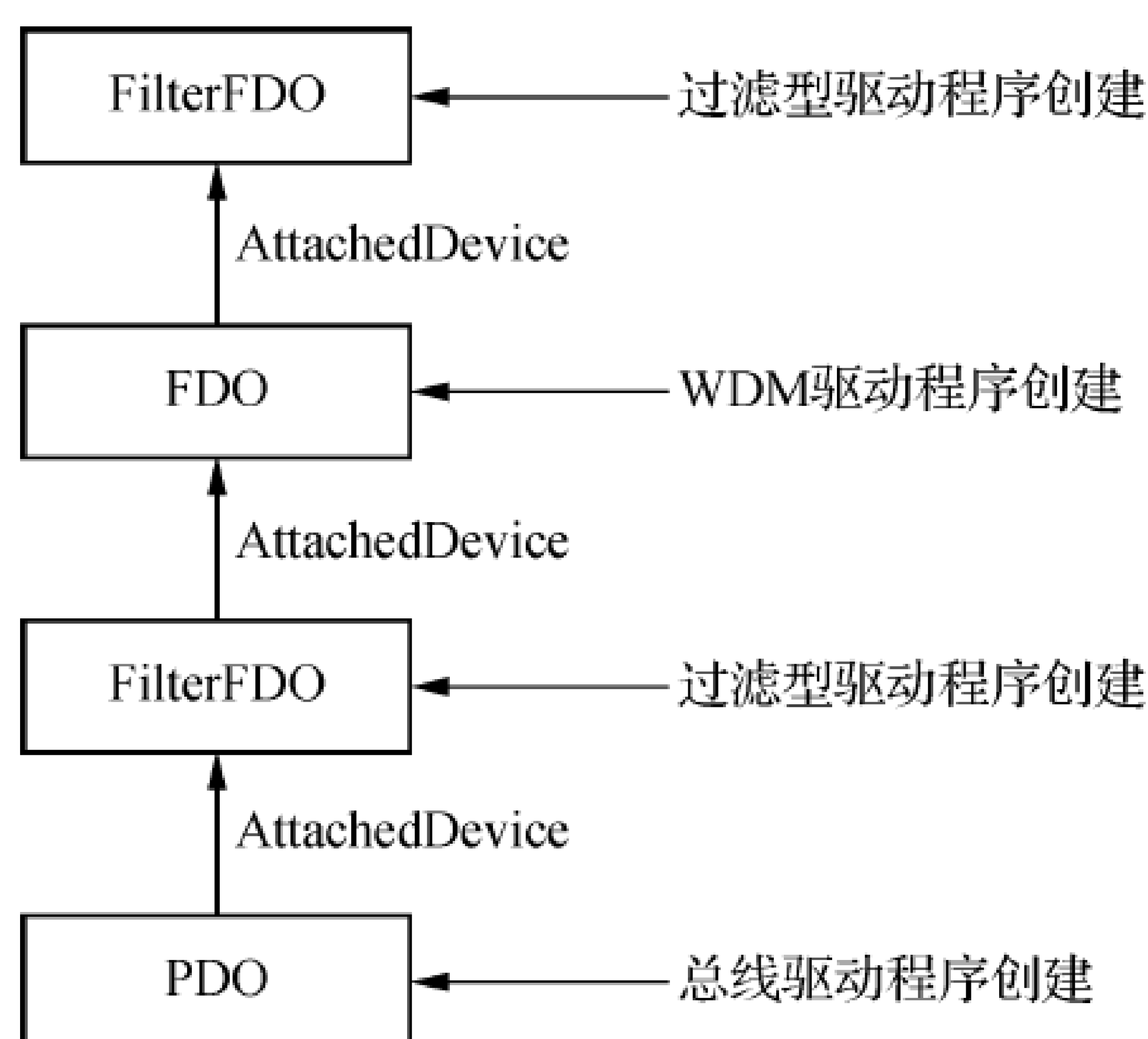


图 16－13　PDO、FDO 与过滤型驱动之间的附加关系

如图 16－14 所示，辅助功能驱动（AFD，负责 Windows 系统中 socket 机制的实现）创建的设备对象就被网络监测软件附加了过滤型驱动的设备。

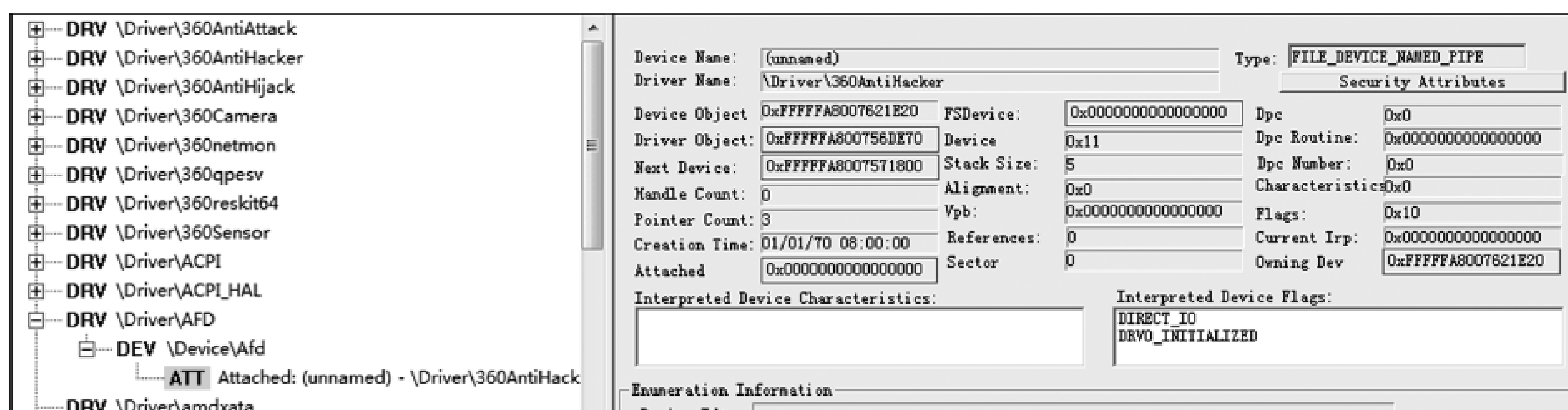


图 16－14　过滤型驱动

在多层设备对象体系下，每个 IRP 都有 I/O 堆栈(IO_STACK_LOCATION)与设备对象相对应，也就是每一层设备对象都对应 I/O 堆栈的一层，因此纵向有几个设备对象就有几层 I/O 堆栈，堆栈中存放了请求的主副功能码，每一层的驱动程序的功能函数也只处理对应层次的 IRP 堆栈，并且每一层堆栈都有权力更改下一层堆栈的功能码。

WDM 源于 NT 式驱动模型，与 NT 式驱动类似，WDM 驱动也包含以下入口函数：

- **驱动程序入口函数 DriverEntry**：与 NT 式驱动一致，WDM 驱动的入口函数也是 DriverEntry，但该入口函数不再涵盖设备创建和附加功能，这部分功能放到了 AddDevice 函数中。由于 WDM 支持即插即用功能，因此也需要在派遣函数数组中设置 IRP_MJ_PNP 元素对应的派遣函数。
- **设备对象创建函数 AddDevice**：该函数是 WDM 驱动所独有的，是 DriverEntry 函数在驱动对象结构的扩展区域 DriverExtension 中设置的，在 DriverEntry 执行完成后该函数被调用。AddDevice 创建功能设备对象(FDO)，并将其附加到总线驱动创建的物理设备对象(PDO)之上。
- **驱动程序卸载函数 DriverUnload**：该函数在 WDM 驱动中的功能非常简单，主要是释放在 DriverEntry 中申请的内存等资源。但是删除设备和删除符号链接的工作不再由它负责。
- **IRP_MJ_PNP 派遣函数**：设备与符号链接的删除是通过 IRP_MN_REMOVE_DEVICE 子功能码对应的处理逻辑实现的。设备被卸载的时候，PNP 管理器向驱动程序发送 IRP_MJ_PNP(子功能码为 IRP_MN_REMOVE_DEVICE)的 IRP，对应的派遣函数会进行设备和符号链接的删除。注意，此时删除的是 FDO 而非 PDO，PDO 是由总线驱动创建的，也理应由总线驱动删除。

16.4　WDF 驱动模型

WDF(Windows Driver Framework，Windows 驱动框架)是 Windows 最新的驱动框架模型，也是一种面向对象的事件驱动机制。微软从 Windows Vista 开始支持 WDF 驱动模型，它目前已成为 Windows 驱动开发的主流选择。

WDF 驱动模型分为内核模式驱动框架(KMDF)和用户模式驱动框架(UMDF)两类，分别对应了 DLL 和 SYS 两种驱动载体形态。与此相对应地，Windows 也将一些低速外设的驱动挪到了用户态空间。在 NT 式驱动模型和 WDM 驱动模型中，驱动程序一般是在内核态空间的，但是从 WDF 驱动模型开始，驱动也可以存在于用户态空间了，设备驱动程序的开发门槛也由此大大降低。

其实，化繁为简始终是人们追求的目标，如果条件允许大可不必把所有驱动都集中于内核态空间，这种“双态并存”也是 WDF 与之前两种模型的显著差异。相较于 WDM，WDF 自己实现了电源管理、PNP 等功能，驱动程序本身不需要对此过多关注，因此开发更简单了。

16.4.1 KMDF

WDF 驱动模型在 Windows 中的具体实现是 Wdf01000.sys 模块，该模块也创建了一个名为 KMDF0 的内核功能设备对象（FDO），如图 16－15 所示，而对应的物理设备对象可见图 16－16。

WDF 继承于 WDM，因此模型本身也分为 FDO 和 PDO 两部分。同时，WDF 模型还具有以下特点：

➢ 向前兼容，WDF 模型兼容 Windows Vista 以前版本的操作系统，也就是兼容这些版本的 WDF 模型的驱动。

➢ 基于对象编程，有一个基本对象，其他诸如驱动对象、设备对象、I/O 请求对象、队列对象等都是基于该基本对象做的派生和扩展，且具有良好的封装和设备驱动接口（Device-Driver Interface，DDI），更加符合面向对象编程的思想。

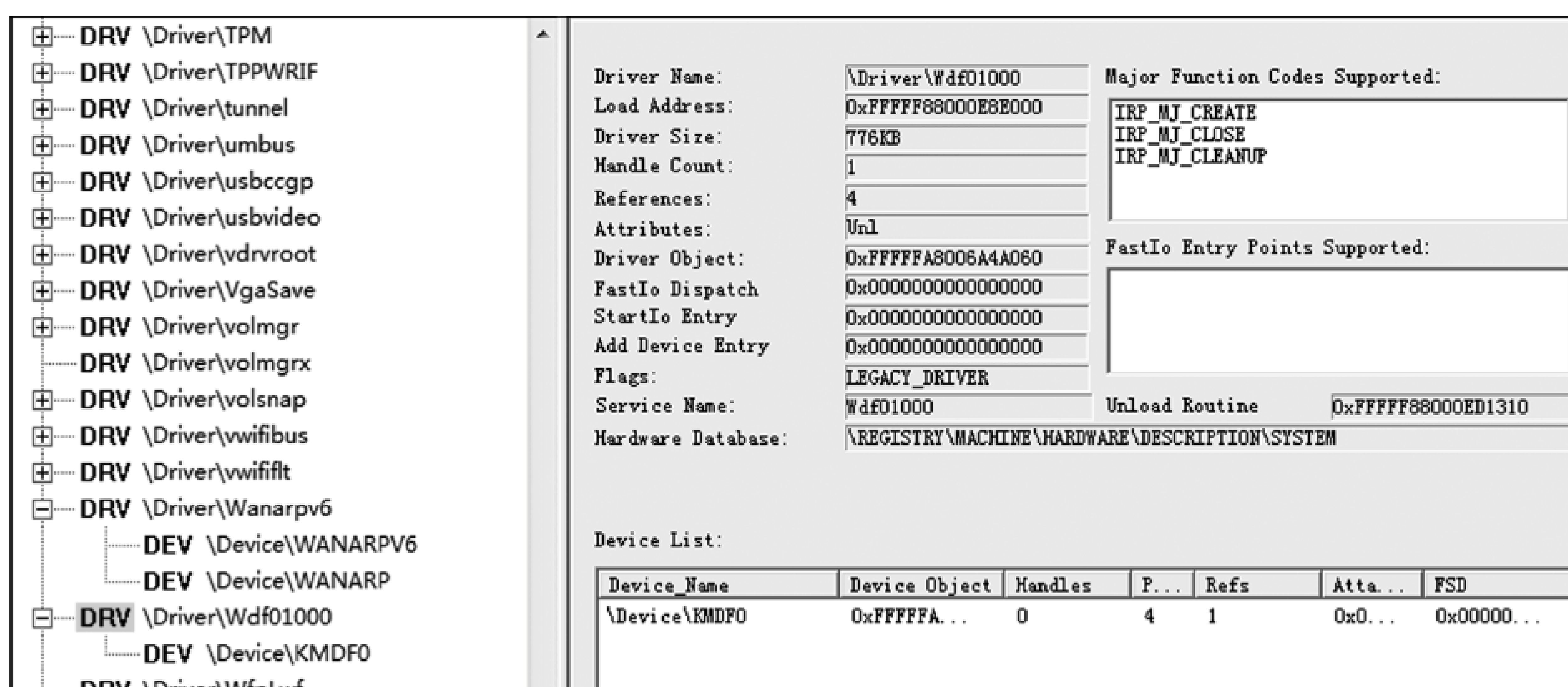


图 16－15　Wdf01000.sys 模块和 KMDF0 设备对象（FDO）

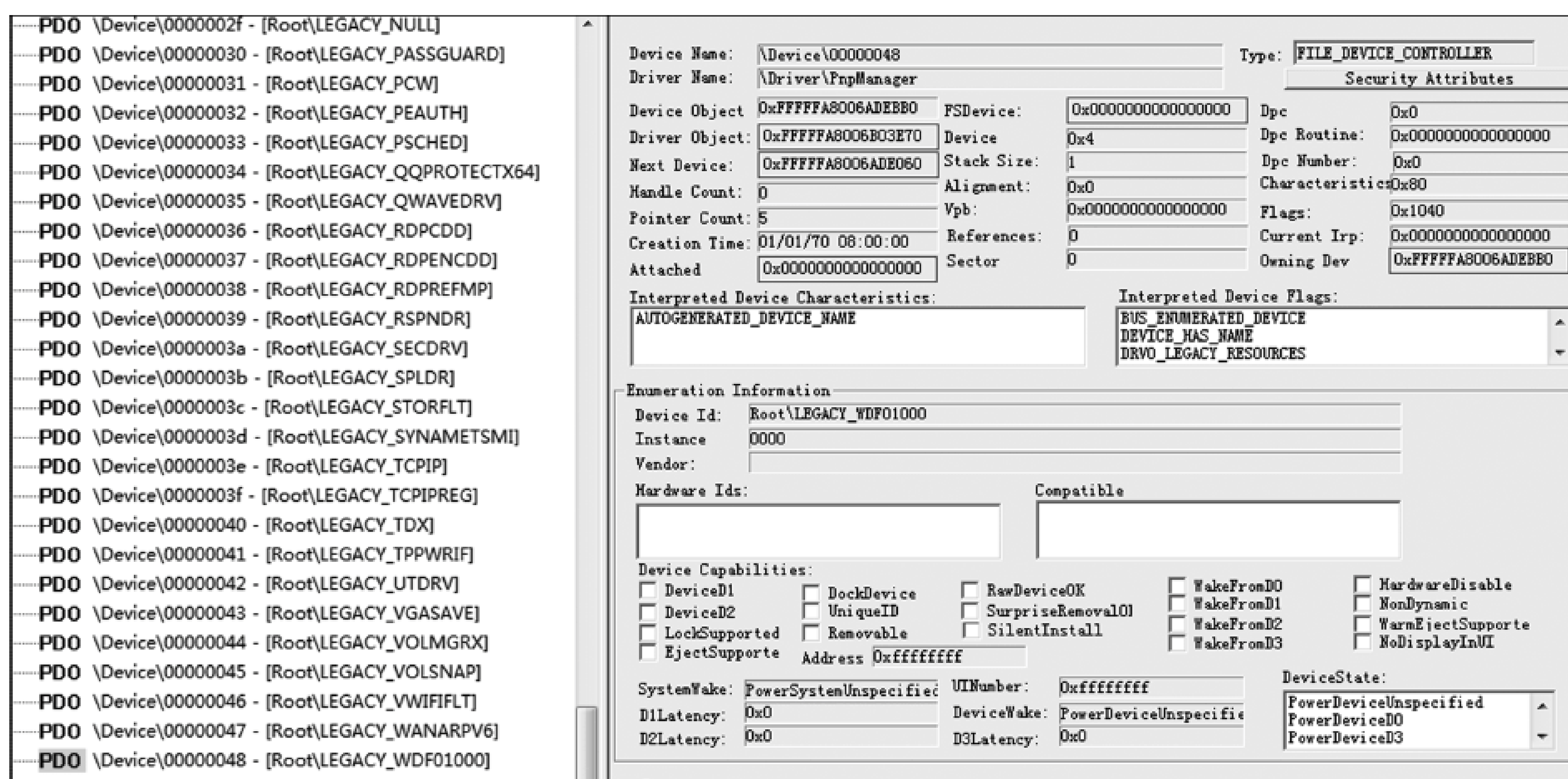


图 16－16　WDF 模型的物理设备对象（PDO）

➢ WDF 驱动模型中的对象一般对应于 WDM 驱动模型中的某数据结构，例如 WDF 驱动模型中的 I/O 请求对象 WDFRequest 相当于 WDM 驱动模型中的 IRP。
➢ WDF 驱动模型通过引用计数的方式管理所有对象的生命周期。
➢ WDF 驱动模型自己实现了即插即用和电源管理功能，开发驱动程序时这部分功能可以由模型代劳。
➢ WDF 驱动模型通过消息队列方式实现了 I/O 请求的串行化。

下面我们以内核模式的 WDF(KMDF) 为例来考察 WDF 驱动模型的结构，其总体架构如图 16－17 所示。

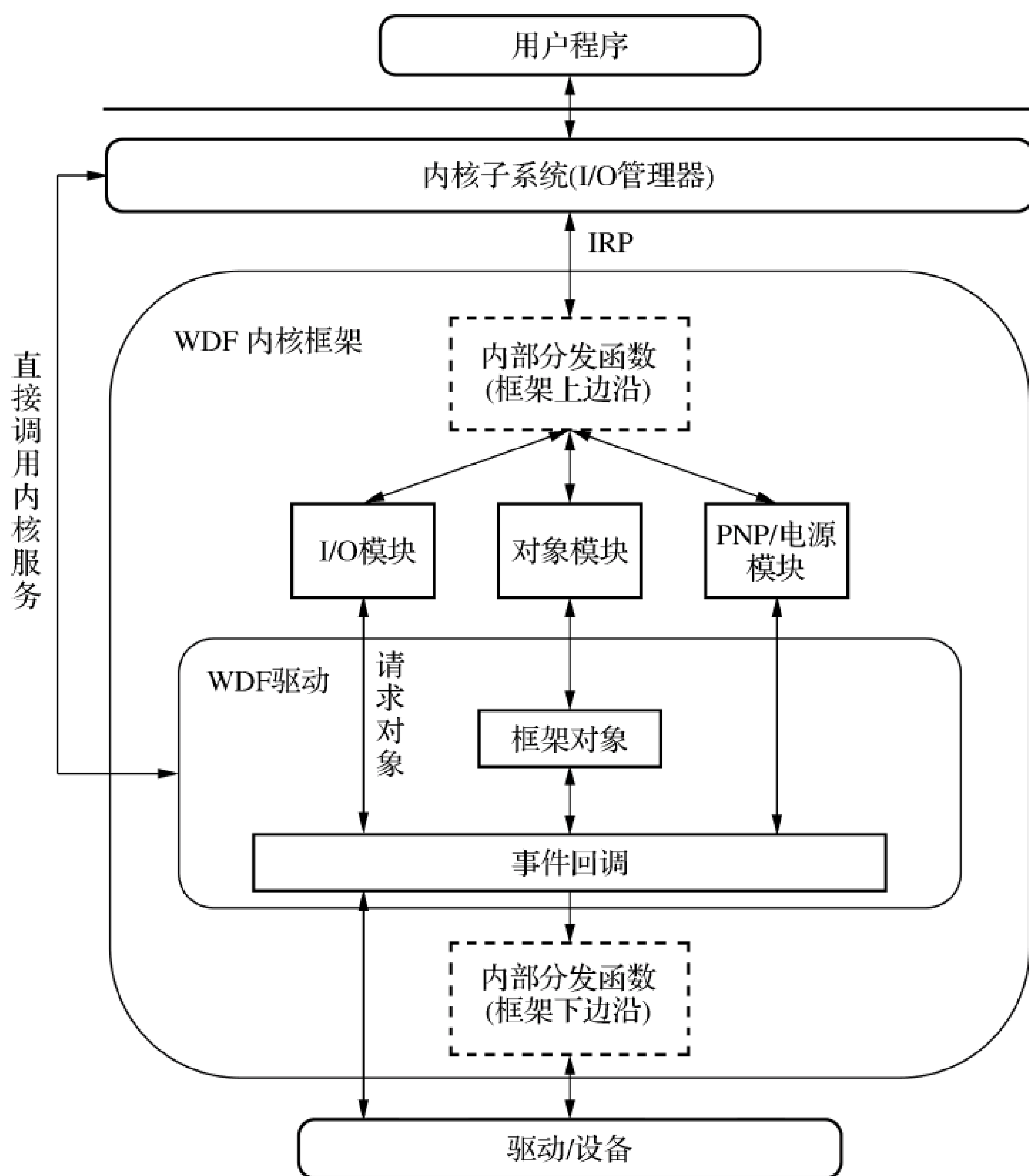


图 16－17　KMDF 的总体架构

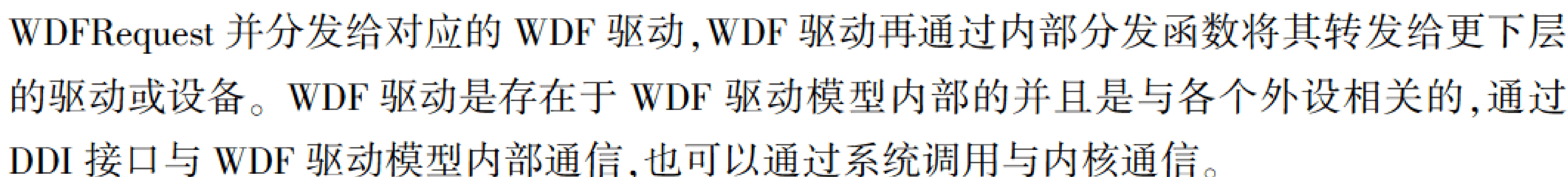
该驱动模型的运行过程是这样的：

(1) 用户程序通过 Windows API 进行 I/O 操作和调用，I/O 管理器通过对应的系统调用将 I/O 操作翻译为 IRP 下发到 WDF 内核框架。注意，I/O 管理器与 WDF 驱动之间的交互依然是通过 IRP。

(2) WDF 内核框架收到 IRP 后通过分发函数将其分发给 WDF 内核框架内的 I/O 相关模块、对象模块、PNP 模块或电源模块。

(3) 上述模块收到 IRP 后将其转化为 WDF 驱动模型内部的 I/O 请求对象结构





WDFRequest 并分发给对应的 WDF 驱动，WDF 驱动再通过内部分发函数将其转发给更下层的驱动或设备。WDF 驱动是存在于 WDF 驱动模型内部的并且是与各个外设相关的，通过 DDI 接口与 WDF 驱动模型内部通信，也可以通过系统调用与内核通信。

（4）当事件发生时（例如读写事件完成）通过回调函数通知到 WDF 驱动，WDF 驱动再通过 WDF 驱动模型的上边沿接口返回给 I/O 管理器。

KMDF 驱动照样也有入口函数 DriverEntry，其作用与 WDM 驱动的入口函数一样。WDF 驱动要向 WDF 驱动模型注册事件回调函数 EvtDriverDeviceAdd，该函数的作用也相当于 WDM 驱动中的 AddDevice，在系统发现新硬件插入时被调用，用于创建设备对象、I/O 队列对象与设备接口等。

WDF 是事件驱动的框架，事件回调函数用于响应驱动程序接收到的 I/O 请求。同时，驱动程序也会创建若干 I/O 请求队列用于持久化和串行化 I/O 请求，这些 I/O 请求队列是与每个 WDF 设备相关的。KMDF 驱动可能还包含即插即用和电源管理的回调函数，在 WDF 处理了即插即用和电源事件以后返回给 KMDF 驱动一个回调通知。

WDF 驱动模型中的对象是具备父子关系的，这不同于面向对象语言中的基类与派生类。WDF 驱动模型中的父对象可以控制子对象，而删除父对象时，父对象会自动将自己的子对象都销毁，因此其生命周期管理也比较方便。在这些对象中，驱动对象是所有 WDF 对象的根对象，代表了加载到系统中的驱动模块，因此无论有多少设备使用该驱动，都只有一个根驱动对象。WDF 模型中对象的种类及其具体的关系图谱如图 16－18 所示。

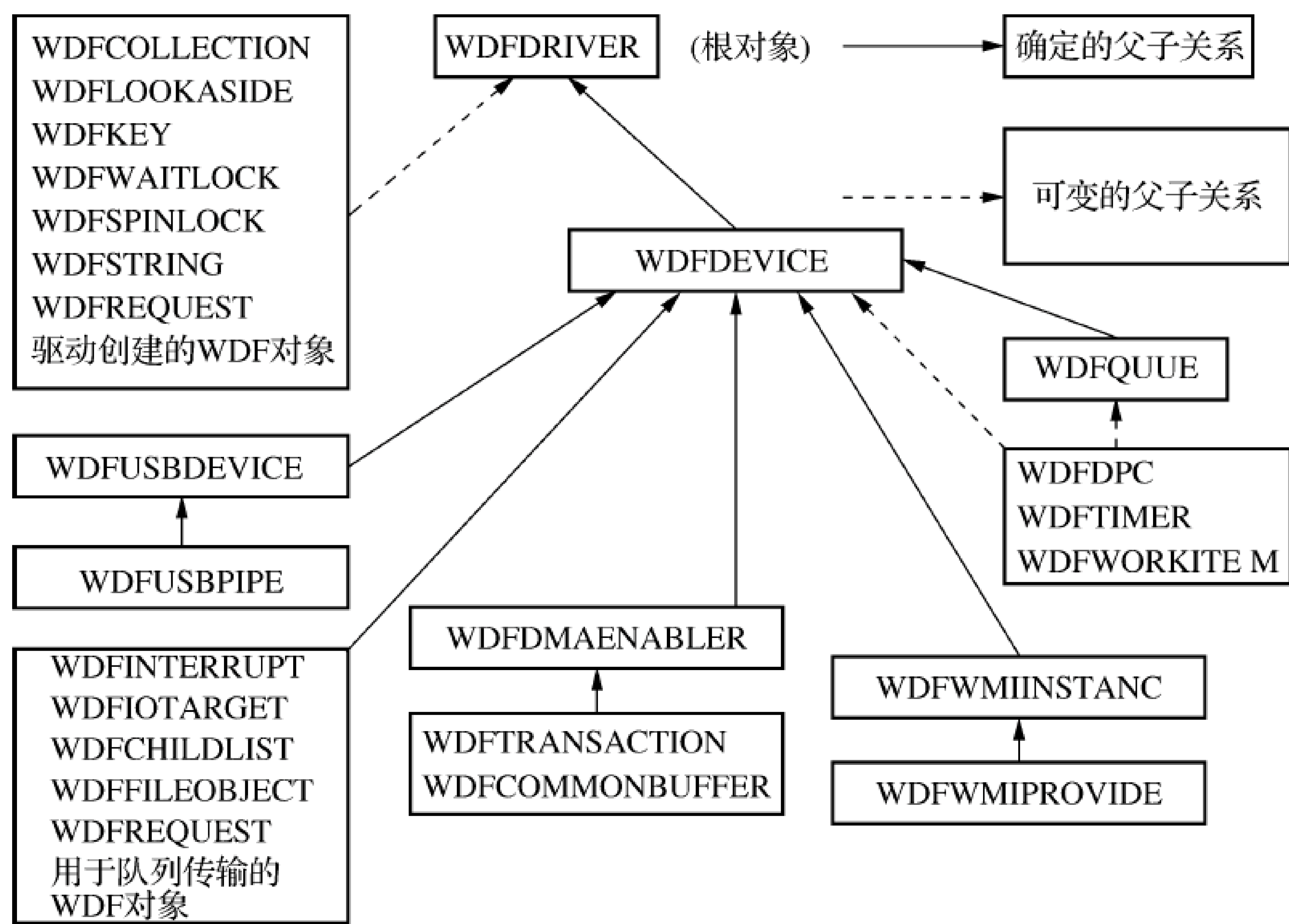


图 16－18　WDF 驱动模型对象的种类及其关系图谱

16.4.2　UMDF

针对用户态模式的 WDF(UMDF),Windows 增加了反射器(reflector)机制,用于将已经“陷入”内核态空间的 IRP 反射回用户态空间的 I/O 驱动进程。增加反射器是由于 Windows 固有的驱动框架的限制造成的,因为历史上驱动程序和 IRP 总是处于内核态空间的,I/O 管理器在内核态空间生成 IRP 后不需要跨空间即可直接下发给驱动程序,但现在要把驱动程序安装在用户态空间,而 I/O 管理器仍然滞留于内核态空间,因此要把代表 I/O 请求的 IRP 下发给驱动程序就必然存在一个跨空间的问题,也就是要从内核态空间跨越到用户态空间。反射器就是用于这个跨空间操作的。

反射器的使用也非常简单:用户态进程调用 Windows API 向 I/O 管理器发送 I/O 请求,内核态的 I/O 管理器将此请求翻译成 IRP 后反射回用户态的 I/O 服务进程,该服务进程就是我们说的 UMDF 模型驱动进程。反射过程如图 16－19 所示。

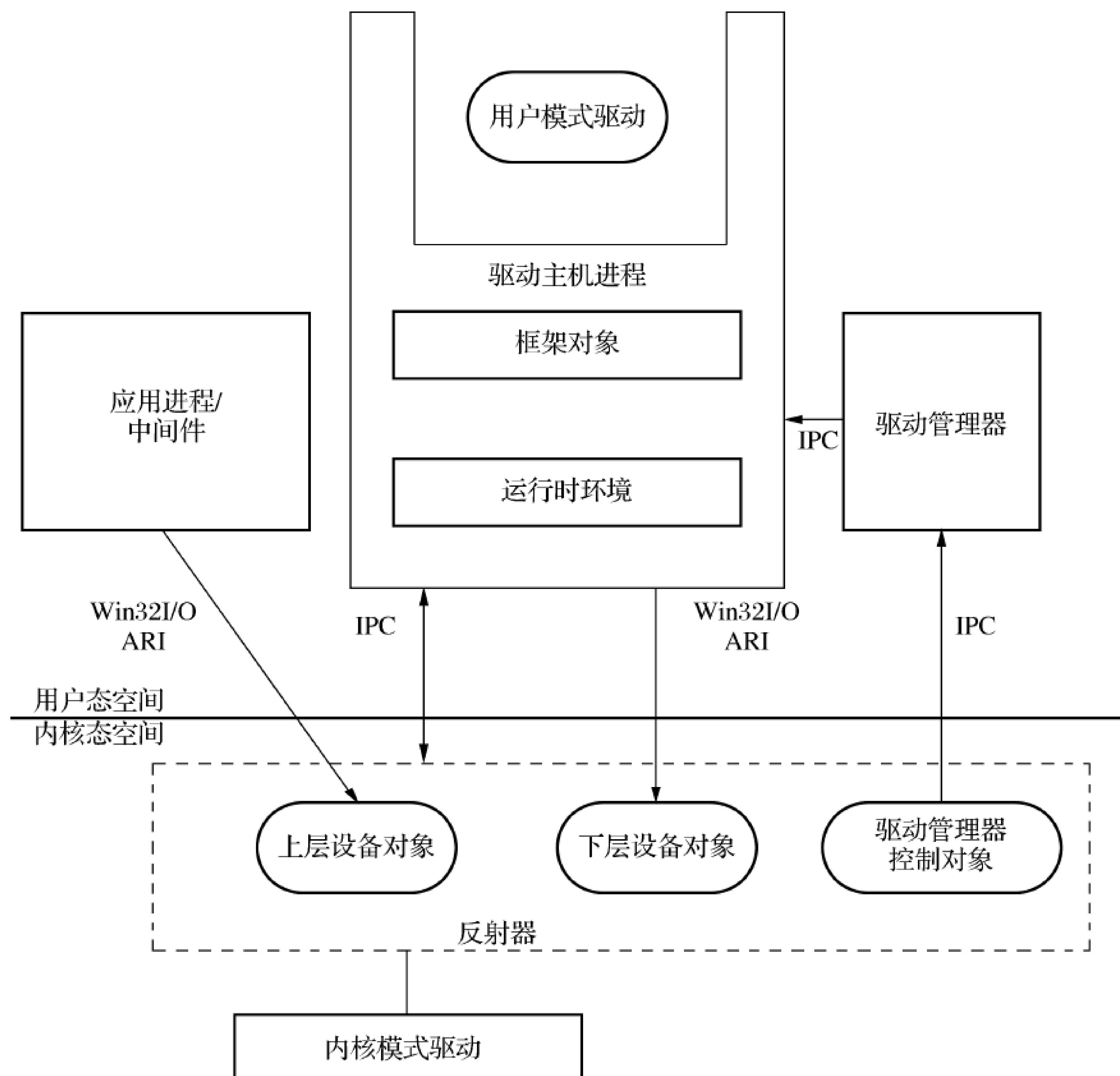


图 16－19　UMDF 的反射过程

在图 16－19 中,驱动主持进程作为驱动管理器的子进程,为用户模式驱动提供运行环境,相当于用户模式驱动的容器。一个驱动管理器可以创建和管理多个驱动主持进程的实

例，一个驱动主持进程只能运行一种设备的用户模式驱动及其过滤驱动（驱动栈）。各个应用进程/中间件通过 I/O 管理器和反射器将请求转发到对应的驱动主持进程，驱动主持进程再将请求转发到对应的用户模式驱动设备堆栈的最顶层设备。用户模式驱动可以直接结束请求，也可以将请求转发到反射器，由反射器转发到内核模式驱动中继续处理。

由于驱动程序处于用户态空间，因此 UMDF 驱动具有以下优点：

- 加固了操作系统的稳定性。运行于内核态空间的驱动一旦崩溃，就会产生 BSOD（死亡蓝屏），整个系统必须重新启动才能恢复正常。但运行于用户态空间的进程仅能访问用户态空间，其崩溃最多影响的是本进程而不会影响到整个系统中其他的进程。
- UMDF 驱动无需考虑运行级别、线程上下文等复杂问题，并且使用用户态调试器即可调试，开发门槛更低。

UMDF 驱动一样也提供即插即用、电源管理、输入输出等功能，并且也可设置 I/O 队列，但不能处理中断、不运行 DMA 机制，也不能自由使用内核的种种资源。

UMDF 由一系列协同运行的对象组成，对象的层级关系如图 16－20 所示。用户模式驱动可以创建和使用这些对象，UMDF 也为这些对象定义了若干方法，在 UMDF 中对象和方法都是基于 COM（组件对象模型）的，由于每个接口都继承于 Iunknown，因此默认也支持 QueryInterface、AddRef 和 Release 方法。

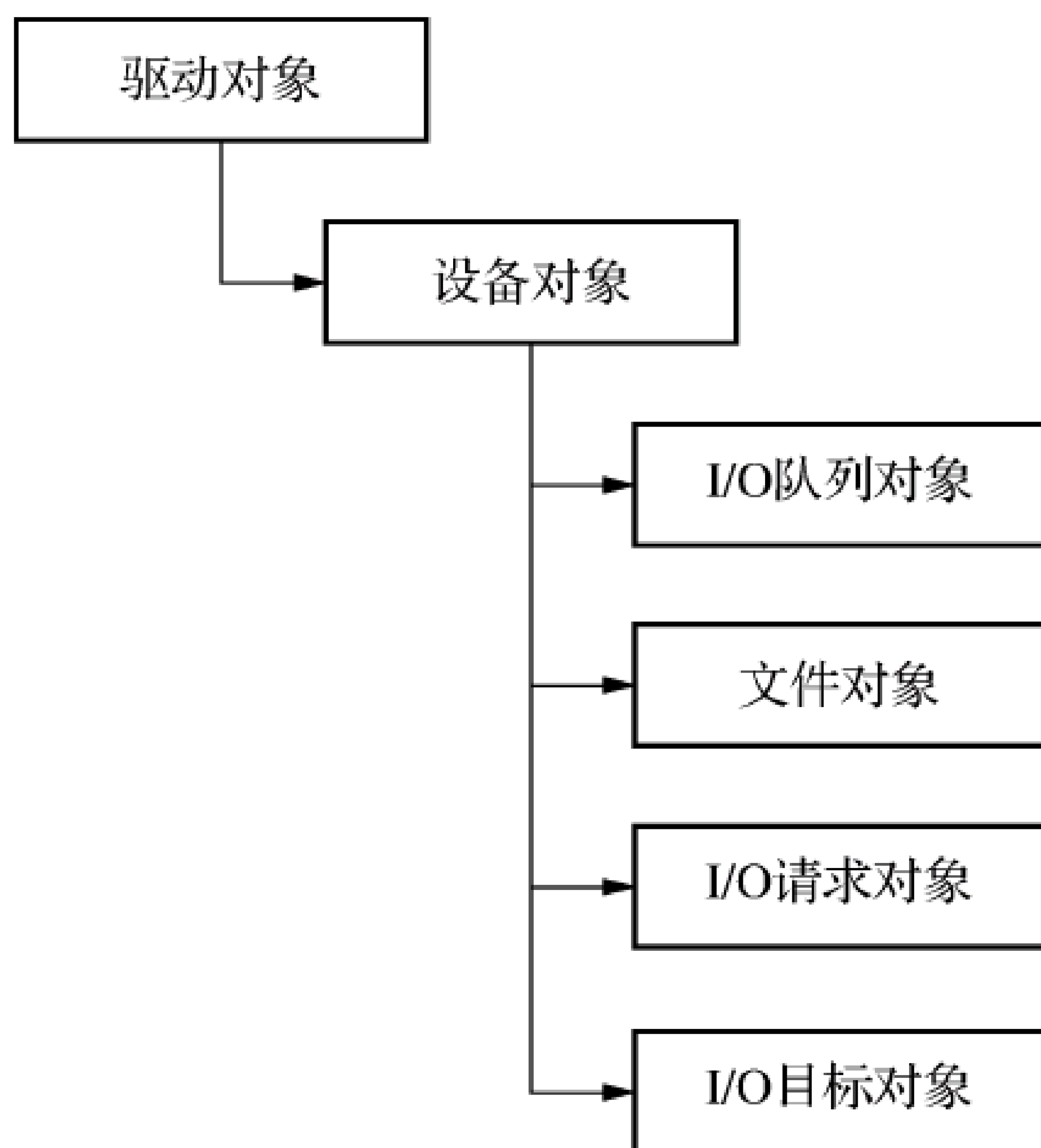


图 16－20　UMDF 对象的层级关系

16.5　PNP 管理器

PNP 管理器负责设备即插即用功能，这个管理器作为执行体的一部分是在系统引导时由 I/O 管理器创建的，创建后 I/O 管理器再通过 PnpInit 方法对 PNP 管理器进行初始化。

即插即用（PNP）的具体执行流程如图 16－21 所示。

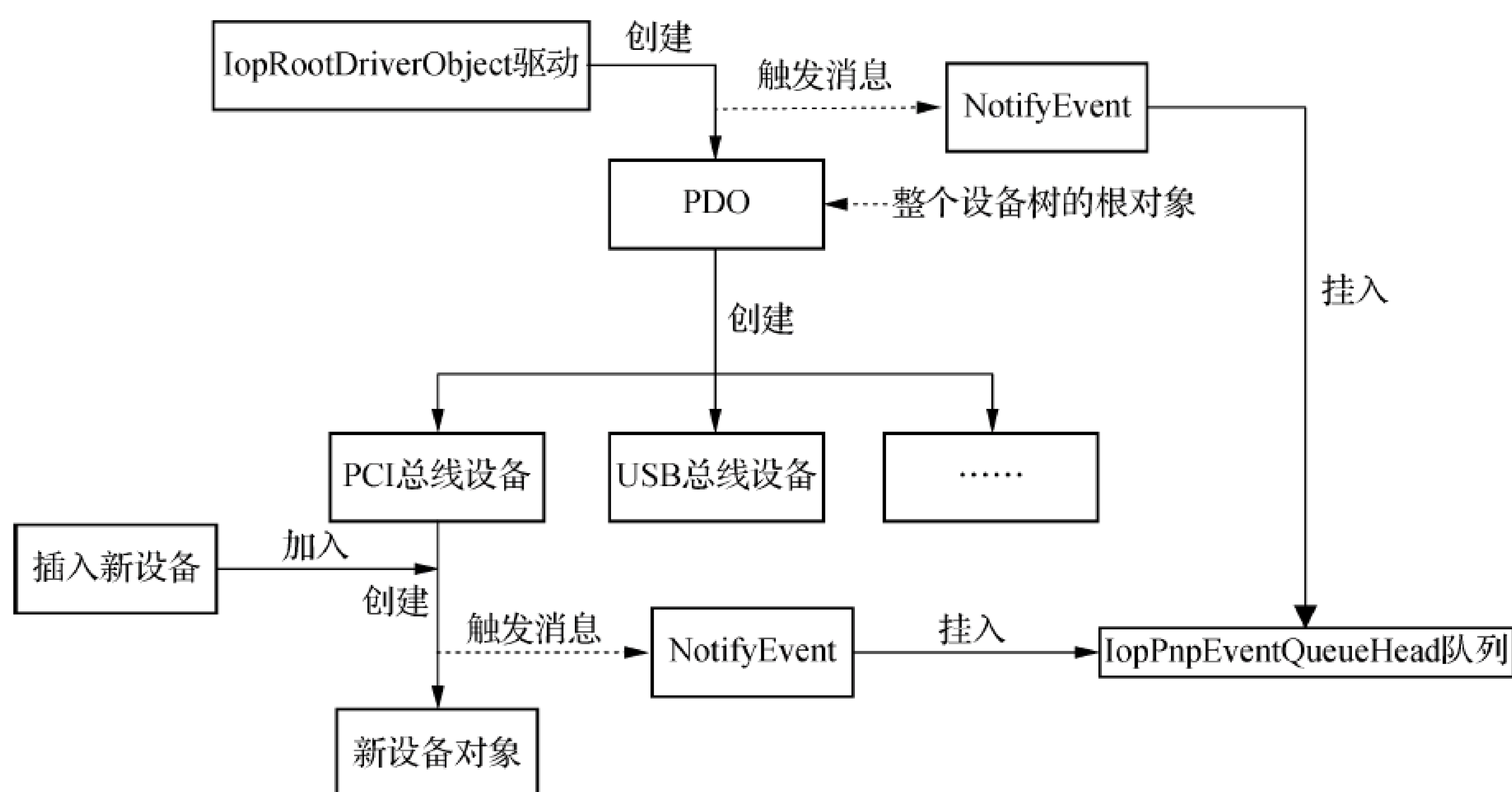


图 16－21　即插即用的执行流程

我们对图 16－21 流程做一下归纳梳理：

➢ IopRootDriverObject 驱动是 Windows 的根驱动，负责创建整个系统中设备树的最上层的根设备对象，这是个物理设备对象。
➢ 创建系统根设备对象时会触发一条事件通知消息——有个物理设备被创建，系统将该消息加入即插即用消息队列（IopPnpEventQueueHead）中。
➢ 随着 PnpInit 函数过程的推进，I/O 管理器创建内核线程 PnpEventThread 处理即插即用消息队列中的消息，即为新创建的设备安装相应的驱动（功能驱动），并且通知系统设备树发生了新的变化。不过上一步中创建的根设备的消息不在此列。
➢ 根设备对象创建好以后再创建 PCI 总线设备对象、USB 总线设备对象和其他总线对象，以支持即插即用的设备枚举等功能。
➢ 当在系统中新插入某设备时，PCI 总线枚举创建新设备对象（PDO），并触发事件通知消息——有个新的物理设备被创建，将该消息加入即插即用消息队列中。
➢ PnpEventThread 线程为新创建的设备安装驱动并通知设备树发生了变化，激活 IopPnpNotifyEvent 事件，从而激活 NtGetPlugPlayEvent 方法。

下面再来看 PnpEventThread 的处理流程：

➢ PnpEventThread 从即插即用消息队列中获取消息，如果是类型为"增加设备"的消息则将其挂入临时队列中。获取消息是通过系统调用 NtGetPlugPlayEvent 实现的，在没有消息时该方法在 IopPnpNotifyEvent 事件上阻塞，以等待新的事件发生。
➢ PnpEventThread 激活 DeviceInstalllistNotEvent 事件，即通知 PNP 管理器内的内核线程 DeviceInstallThread 以安装新的设备驱动，安装完成后 DeviceInstalllistNotEvent 事件阻塞休眠。

从上述流程可知，PNP 框架各组件之间构成了一对"生产者–消费者"的经典关系模型，

它们以即插即用消息队列 IopPnpEventQueueHead 为纽带，如图 16－22 所示。

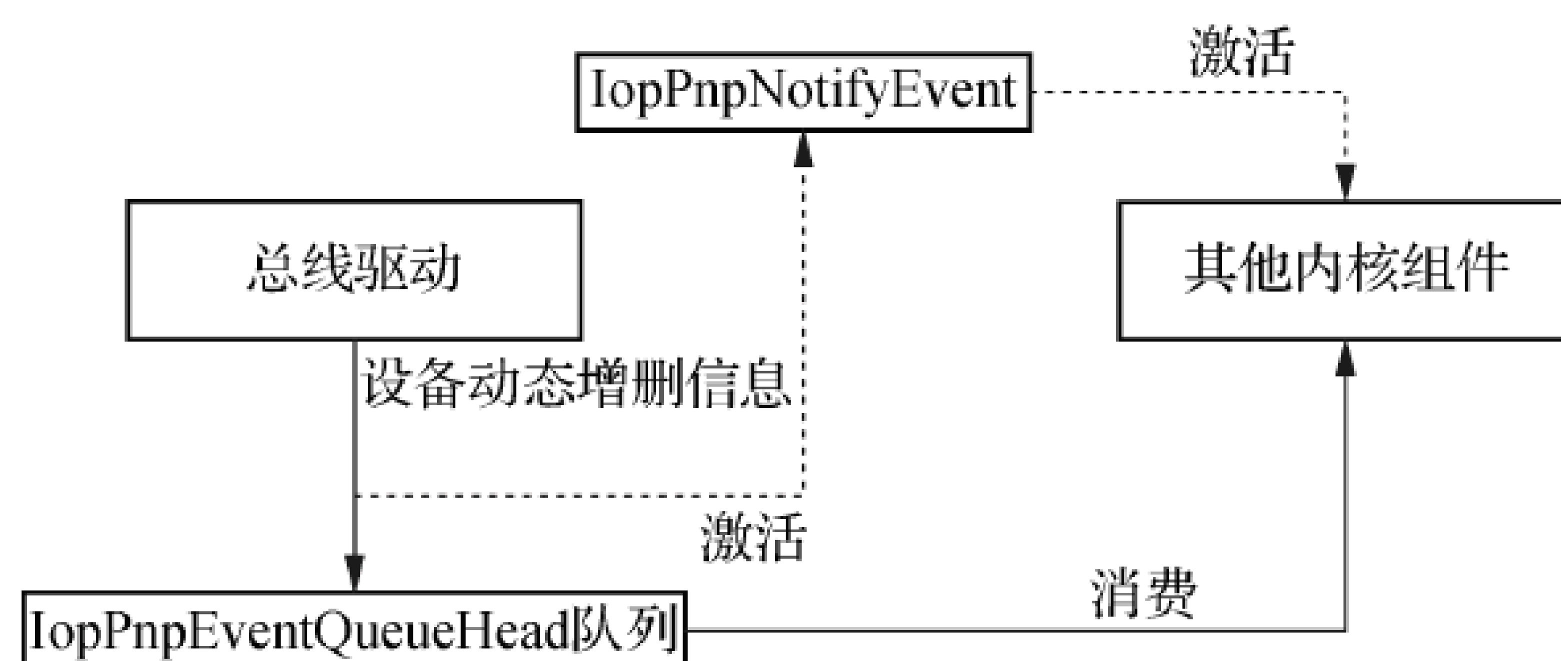


图 16－22　PNP 消息的生产与消费

PnpInit 除了创建根驱动对象和根设备对象，还要创建根设备节点 IopRootDeviceNode。设备节点是设备树的组成结构，用于表示父子、兄弟关系和其他一些关联信息，其数据结构为 DEVICE_NODE，如下所示：

```
typedef struct_DEVICE_NODE
{
    struct_DEVICE_NODE * Parent;
    struct_DEVICE_NODE * NextSibling;
    struct_DEVICE_NODE * Child;
    ......
    PDEVICE_OBJECT PhysicalDeviceObject;
    PCM_RESOURCE_LIST ResourceList;
    ......
}
```

设备节点也与设备对象关联在一起，如图 16－23 所示，根节点以下有两个子节点分别代表了 PNP 设备和 Legacy 设备，而前者还要派生出物理设备节点和功能设备节点，对应了各个驱动的物理设备对象和功能设备对象，这同时也是 WDM 驱动的一般特征。

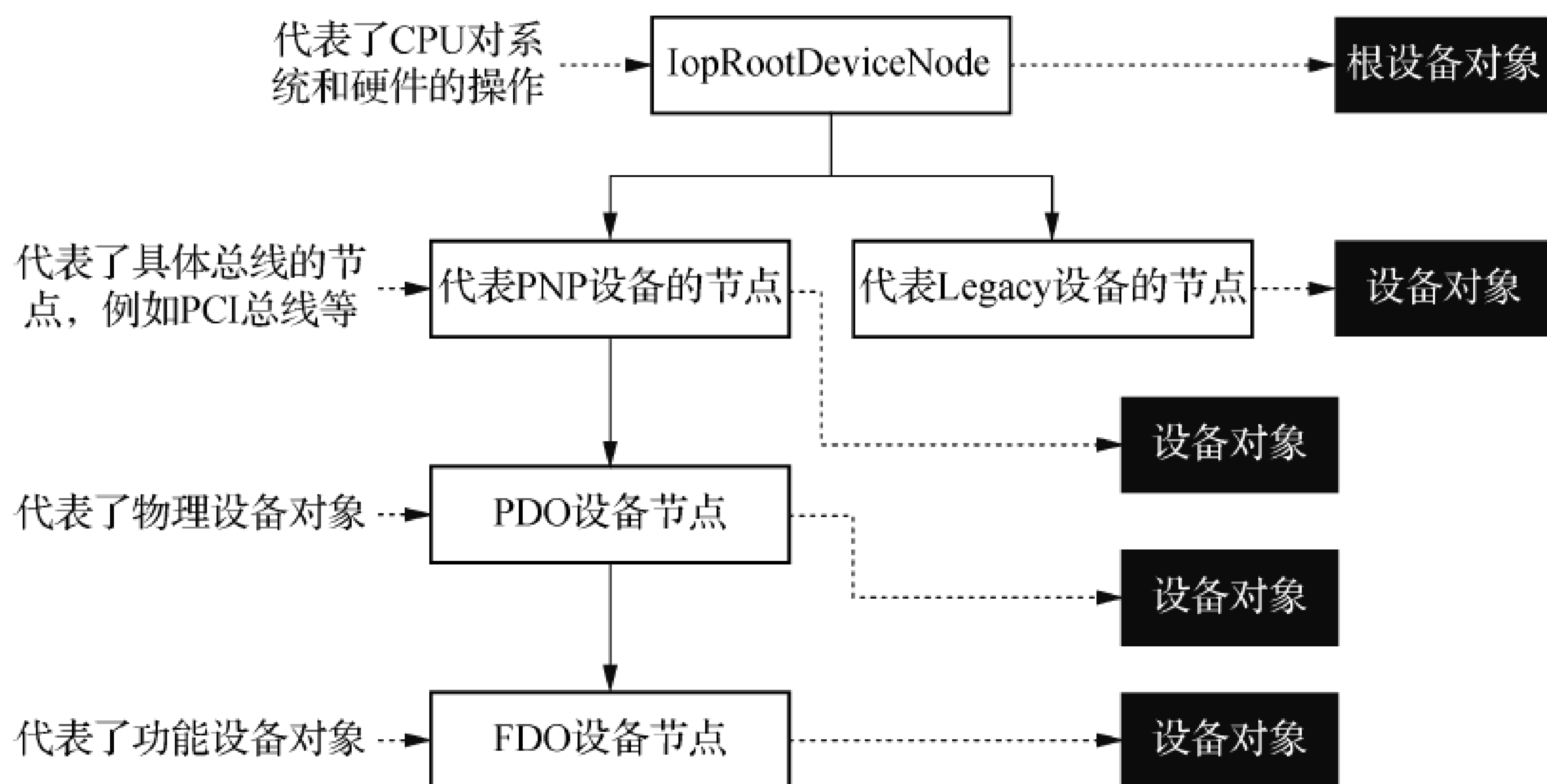


图 16－23　PnpInit 创建的设备节点树

本章小结

本章主要介绍了 Windows 系统下的设备驱动框架。在 Windows 中,驱动框架包含了多种数据结构,例如驱动对象、设备对象等,其信令的下发也不再是传统报文或者 API 的方式。

在 Windows 的发展过程中,先后经历了 NT 式驱动模型、WDM 驱动模型和 WDF 驱动模型三种框架结构,一代比一代先进,一代比一代省力、省时、省电、省脑。特别是 WDF 驱动模型又分为了内核模式 WDF 和用户模式 WDF,为降低驱动程序开发的门槛做了很好的铺垫。

第 17 章　Windows 网络协议栈驱动

本章将按照图 17－1 所示的提纲对网络协议栈驱动的相关技术进行梳理。

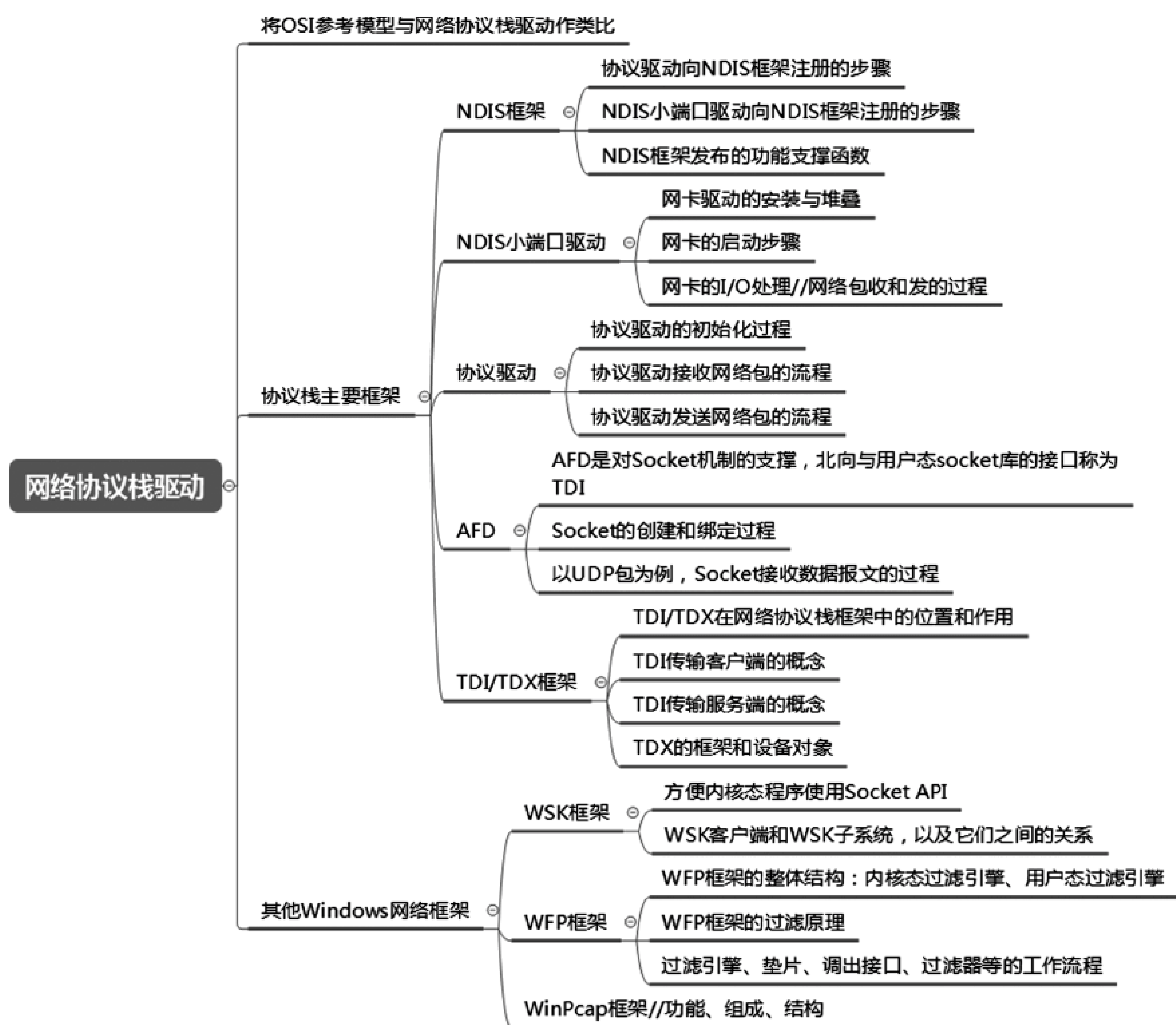


图 17－1　本章提纲

OSI(Open System Interconnect,开放系统互连)参考模型将网络协议栈分为 7 层,从下到上的层次如图 17－2 所示。除物理层外的其余 6 层均在操作系统中实现了相关的功能,其中链路层、网络层和传输层的功能是以内核驱动的形态存在于 Windows 系统中的,而另外三层则一般以用户态应用支撑库的形态予以实现。我们在本章只讲述以内核驱动形态存在的下三层的协议栈,这三层也被称为 Windows 网络协议栈驱动。

网络协议栈驱动是实现 OSI 参考模型中若干层网络功能的驱动,其体系结构如图 17－3 所示。可以看出,协议栈驱动的层次大致是按照 OSI 参考模型的层次来划分的。

TDI(Transport Driver Interface,传输驱动接口)规范将协议栈划分为两层,TDI 规范以上

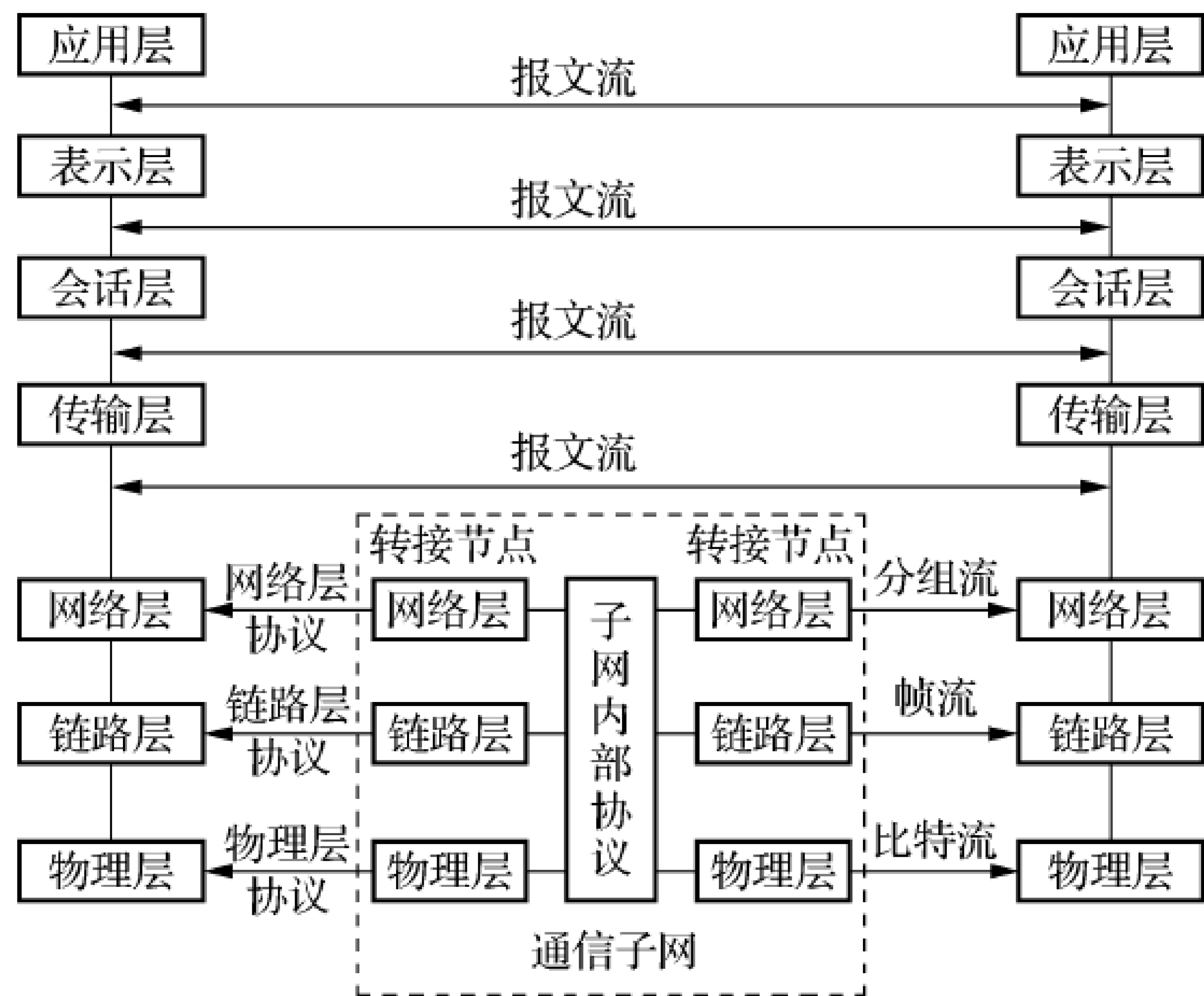


图 17－2　OSI 参考模型

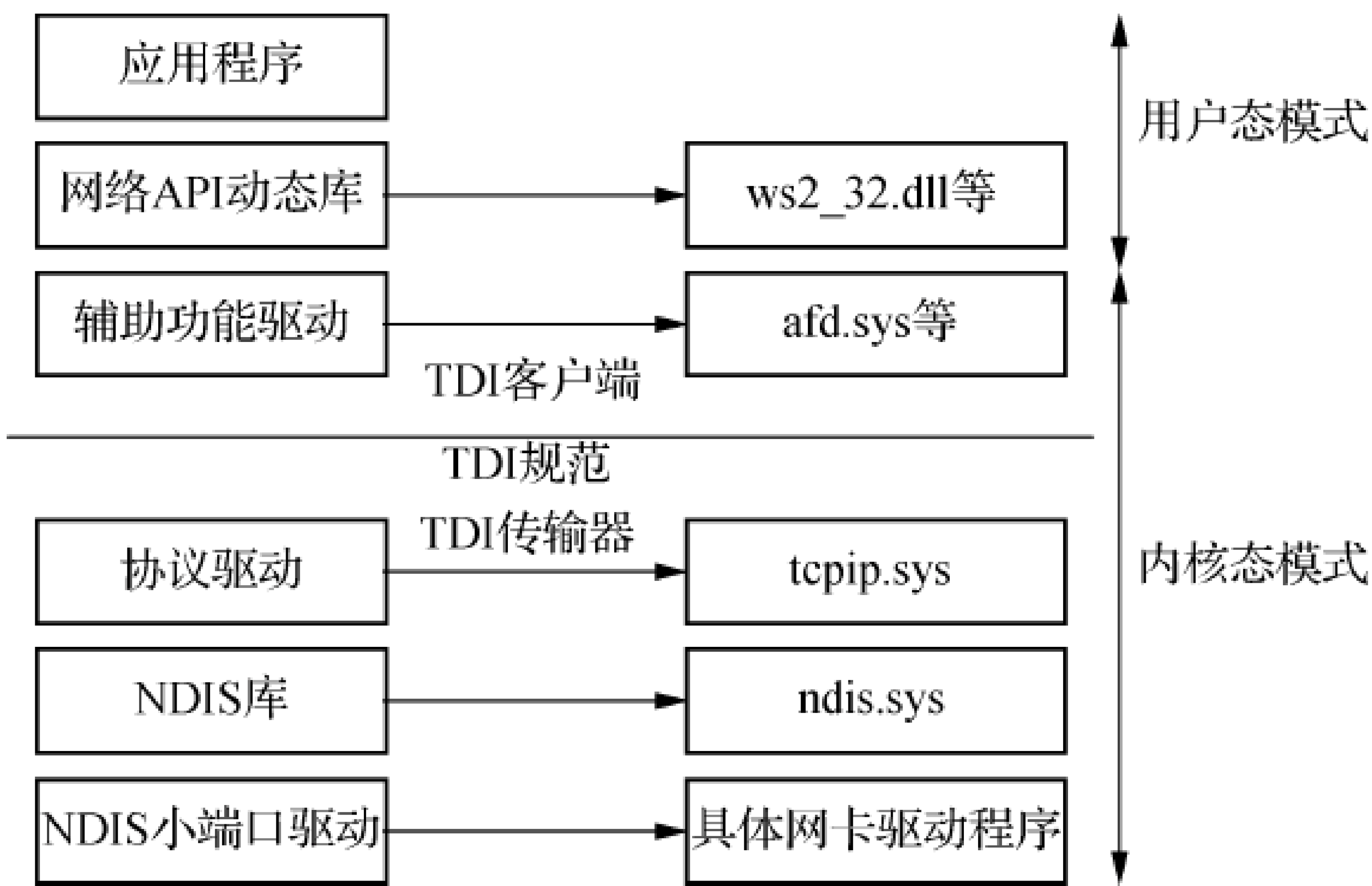


图 17－3　Windows 网络协议栈体系结构

的部分称为 TDI 客户端，以下的部分称为 TDI 传输器。TDI 是连接辅助功能驱动和协议驱动的接口，实现了协议驱动与辅助功能驱动的解耦和对接。

图 17－4 中的 NDIS 小端口驱动就是具体的网卡驱动，这一般是由网卡厂商来提供的，例如本机中存在的 Intel 以太网网卡驱动 eld62x64. sys、无线网卡驱动 netwsw04. sys 和 vwifibus. sys 等。NDIS 小端口驱动负责具体型号网卡的控制和操作，大致对应了链路层的下半部分（MAC 子层——介质访问控制子层）。

驱动名	驱动类型	基地址	大小	驱动对象	驱动路径	文件厂商
e1d62x64.sys	一般驱动	0xfffff88008148000	0x80000	0xfffffa80077cca10	C:\Windows\system32\D...	Intel Corporation

图 17－4　NDIS 小端口驱动 eld62x64. sys

如图 17－5 所示，以太网 MAC 芯片的一端连接到计算机 PCI 总线，另外一端连接到以太网 PHY 芯片上，它们之间是通过 MII（介质独立接口——IEEE 802. 3 定义的以太网行业标准）对接的。以太网 PHY 是物理接口收发器，它实现了物理层功能。IEEE 802. 3 标准定




义了以太网 PHY,包括 MII/GMII(介质独立接口)子层、PCS(物理编码)子层、PMA(物理介质附加)子层、PMD(物理介质相关)子层。

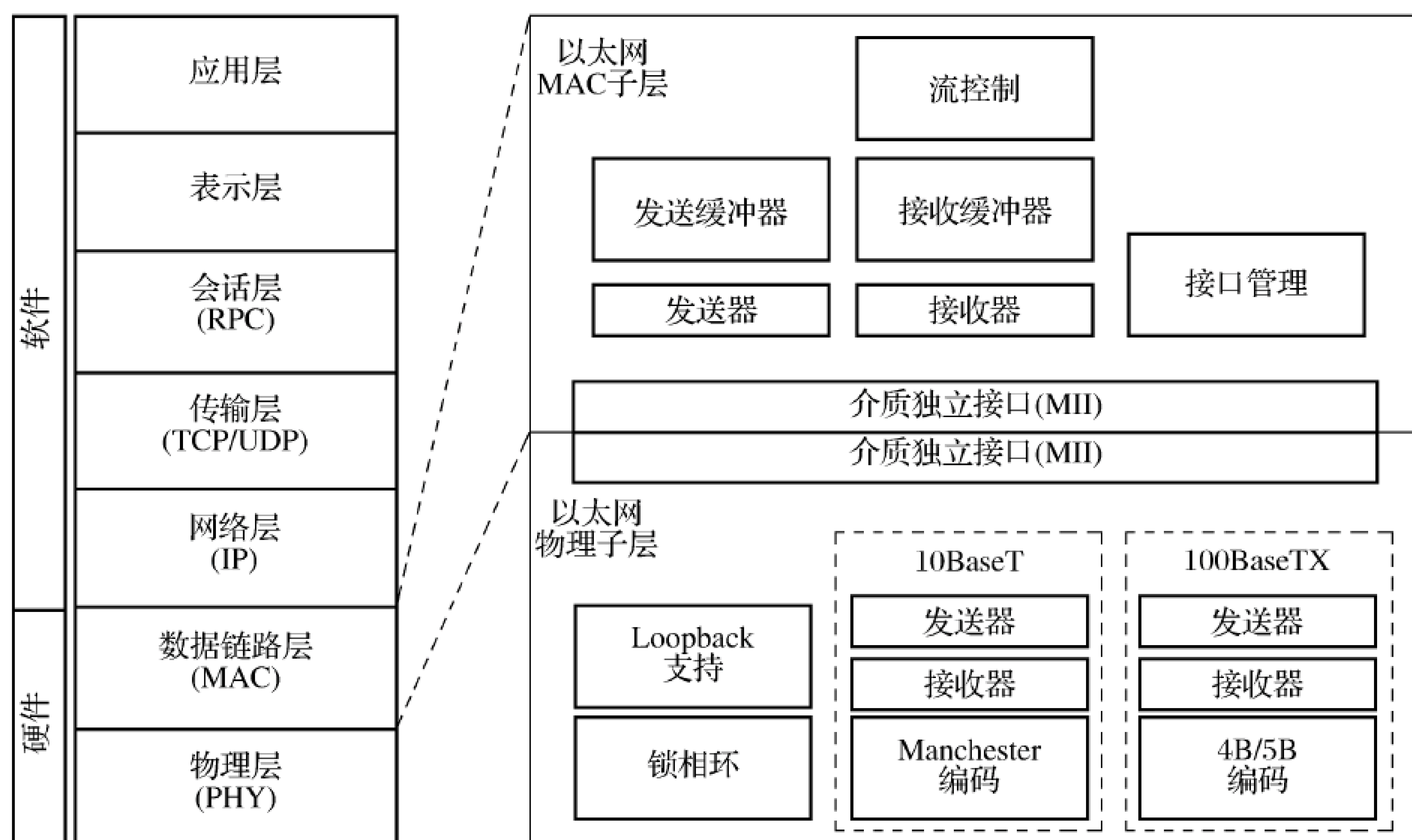


图 17－5　链路层与物理层的硬件体系结构

那 NDIS 又是什么呢? NDIS (Network Driver Interface Specification,网络驱动接口规范)是由微软与 3COM 在 1989 年联合制定的。NDIS 库是网络协议栈驱动中最重要的模块,起着承上启下的作用,它允许上层的协议驱动和下层的 NDIS 小端口驱动(网卡驱动)通过上下边沿接口向它注册回调函数指针,以便打通协议驱动和网卡驱动之间的联系,同时也提供了 NDIS 风格的系统调用封装接口,如图 17－6 所示。NDIS 不代表 OSI 参考模型中的任何一层,但 NDIS 将操作系统实现的协议栈功能与各厂商实现的物理网卡功能关联了起来,并实现了协议栈驱动与物理网卡驱动的双向无关性:只要遵循 NDIS 框架和接口规范,协议驱动和网卡设备驱动都可以由不同的开发者提供,甚至我们自己 DIY 的一个协议驱动也可以挂接进来。NDIS 将在后面的章节中专门介绍。

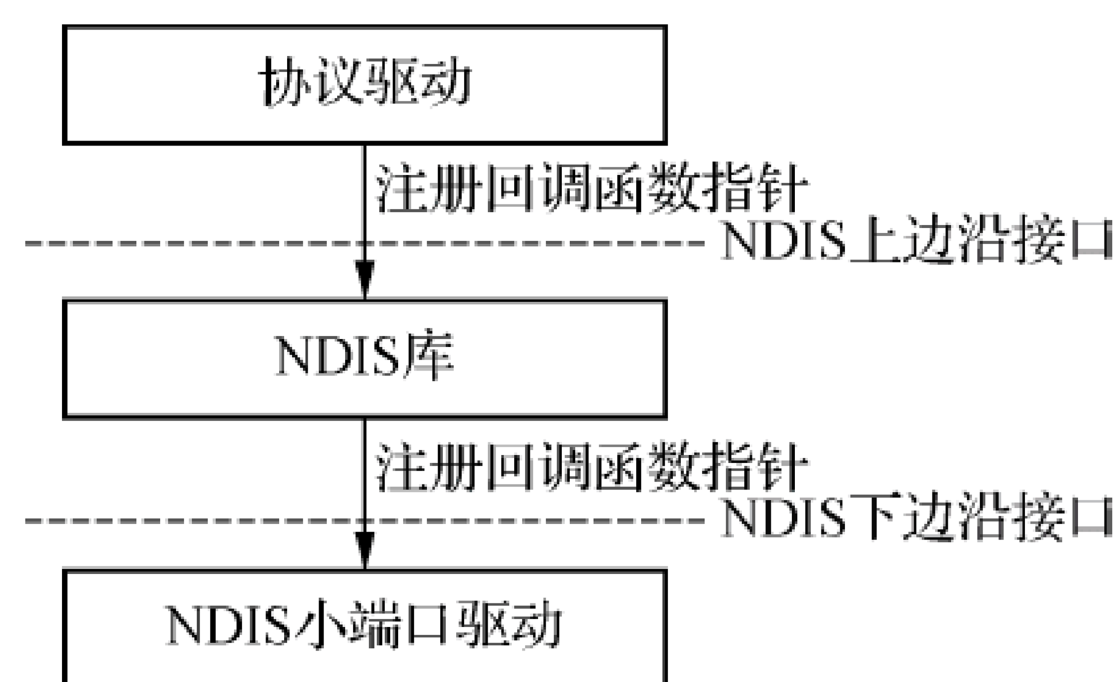


图 17－6　NDIS 上下边沿接口及回调注册

Windows 中的协议驱动就是指 tcpip. sys,这是网络协议栈驱动的核心模块,协议栈中链路层的 MAC 子层、网络层、传输层都是由协议驱动实现的,可以说协议驱动是网络协议栈功能的集大成者。当然 tcpip. sys 是由微软实现的,由于 TDI 和 NDIS 框架的存在,协议驱动也可以由第三方实现,只要在上下边沿分别遵循 TDI 和 NDIS 接口就行了。

辅助功能驱动(Ancillary Function Driver, AFD)用于实现 Socket 机制,具体承载模块是

afd. sys。我们在应用软件中使用的网络库(ws2_32. dll 等)都是通过调用 AFD 进而与协议驱动打交道的。对于 AFD 后文也会有详细介绍。

在 Windows 系统中 AFD 及其下层的驱动都是内核态驱动。AFD 之上的网络 API 接口就是用户态动态库了,Windows 对这一部分已经做了很好的封装,并且许多通信框架又基于这些网络 API 做了二次封装,使调用更加简洁、功能更加强大、体系更加框架化,后续章节会有介绍。

17.1　NDIS 框架

NDIS 是协议栈驱动中的重要框架,连通了协议驱动与 NDIS 小端口驱动,同时也提供了很多功能性函数供 NDIS 框架成员调用,因此我们可以将其看作协议驱动与 NDIS 小端口驱动的隔离层。我们在网络包检测和过滤的时候经常在 NDIS 框架上挂钩相应的过滤型驱动程序,作为旁路驱动检测数据包。而防火墙和虚拟网卡的实现也都有赖于 NDIS 框架,并由 NDIS 框架中间层驱动实现。甚至我们在后文要讲述的 WinPcap 框架(wireshark 抓包工具的主要框架)也是基于 NDIS 框架实现的。由此可见 NDIS 框架在网络安全和网络分析领域有着多么广泛的应用。

NDIS 是个比较复杂的框架,把它看成一个“包装器”(Wrapper)可能更贴切。从图 17 - 7 可以看出,NDIS 框架提供了对 NDIS 小端口驱动调用 HAL(Hardware Abstraction Layer,硬件抽象层)的支持、NDIS 小端口驱动与中间层驱动/协议驱动的互通机制、中间层驱动与协议驱动的互通机制等。其中,中间层驱动也可以由多个驱动堆叠起来构成,在流量负载均衡体系中,这是一种常用的机制和方法。因此,NDIS 不是某几种驱动之间的通信框架,而是一个也包含了系统调用、HAL 访问支持在内的全方位框架。不过 NDIS 中间层驱动不是我们考察的重点,本章主要介绍 NDIS 框架对协议驱动与 NDIS 小端口驱动的支持。

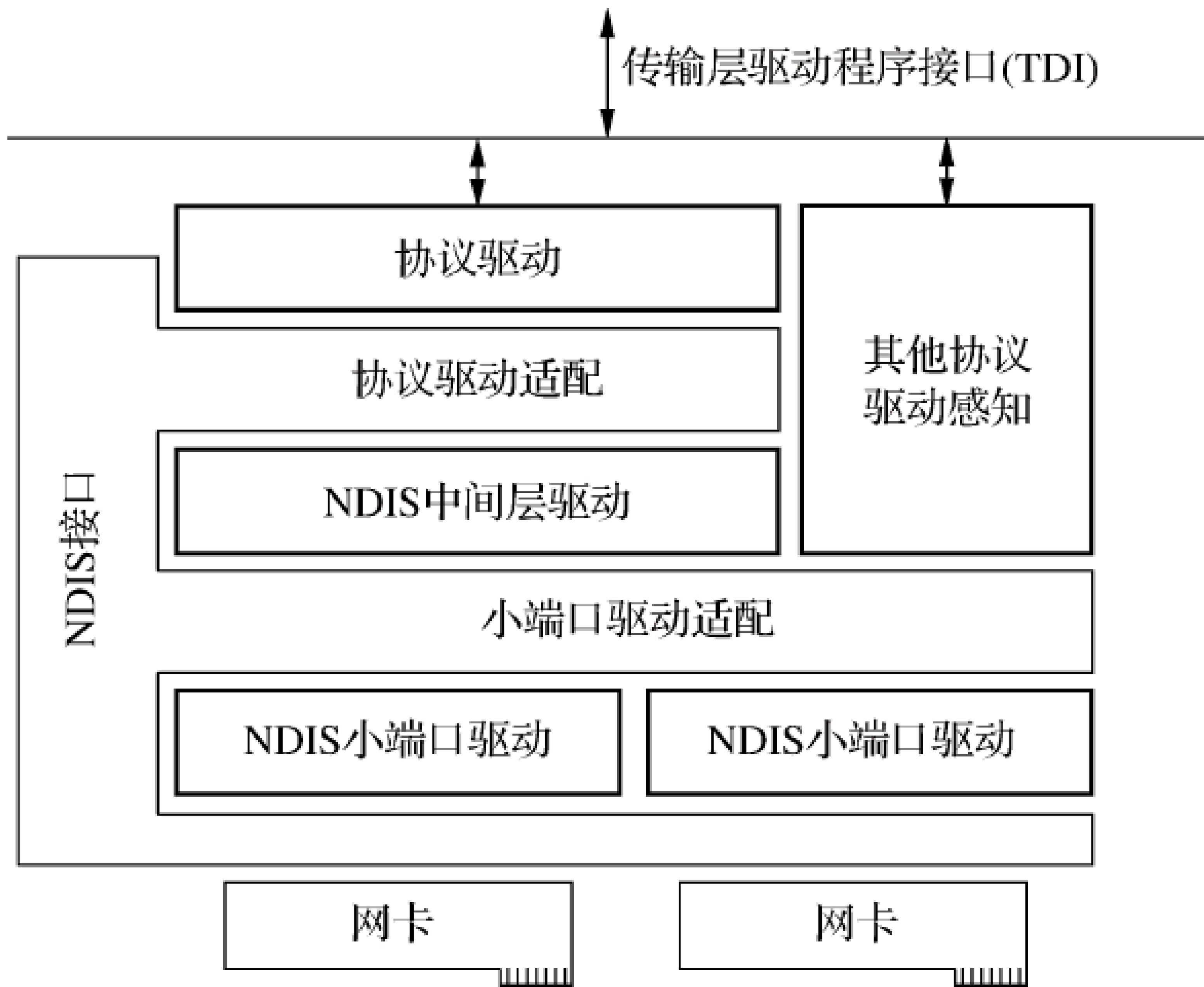


图 17 - 7　NDIS 整体框架

协议驱动与 NDIS 小端口驱动在初始化的时候都要向 NDIS 框架注册，如图 17－8 所示，我们将协议驱动和 NDIS 小端口驱动的注册步骤对比着来介绍：

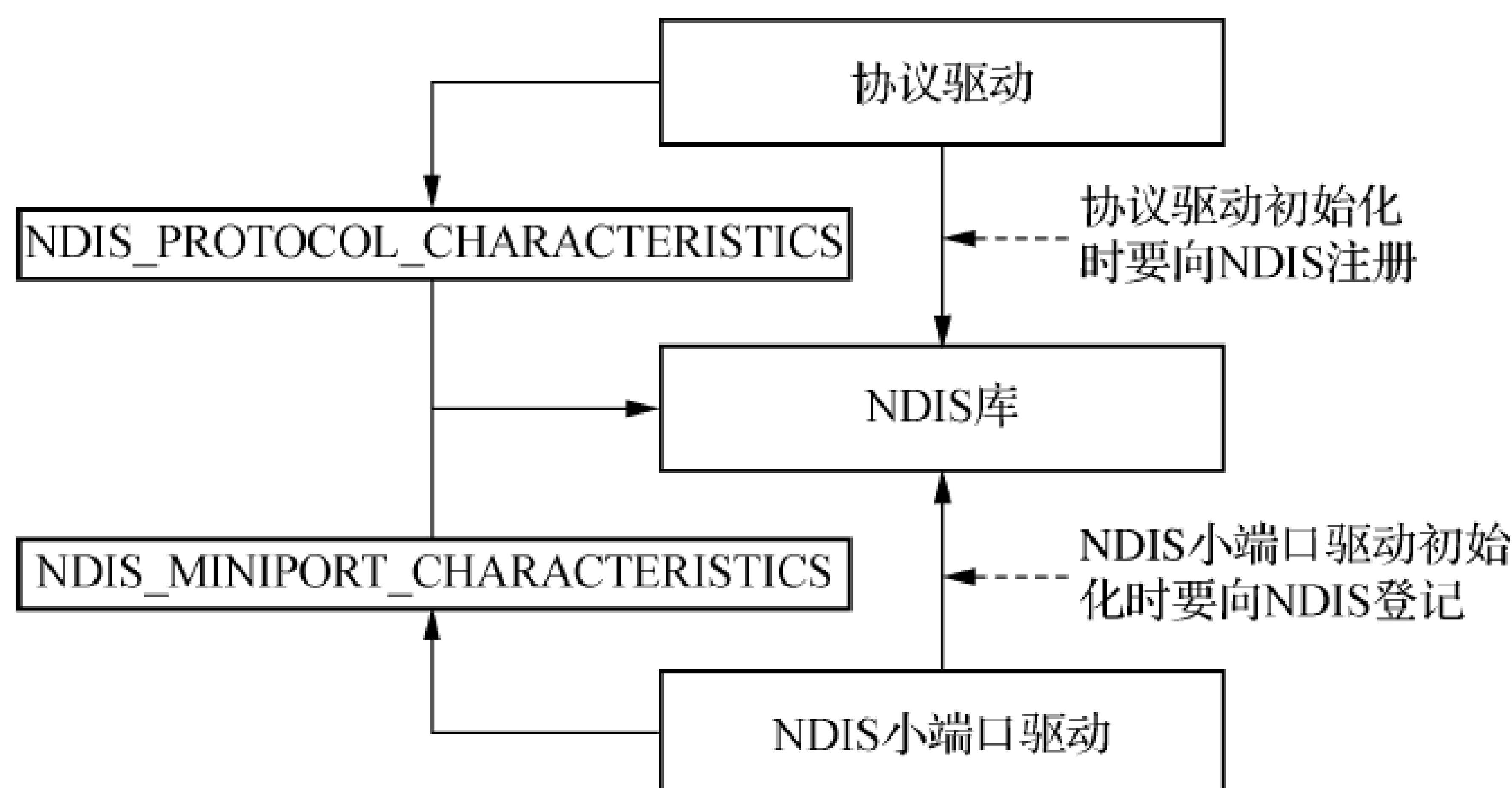


图 17－8　协议驱动与 NDIS 小端口驱动向 NDIS 注册的步骤

- NDIS 框架的加载要先于协议驱动和 NDIS 小端口驱动的加载。
- 协议驱动通过 NDIS 框架的 **NdisRegisterProtocol** 方法向 NDIS 框架注册；NDIS 小端口驱动通过 NDIS 框架的 **NdisRegisterMiniport** 方法向 NDIS 框架注册。
- 使用 NdisRegisterProtocol 注册时，协议驱动使用 **NDIS_PROTOCOL_CHARACTERISTICS**（协议特征块）数据结构携带协议驱动的回调函数指针向 NDIS 框架登记。协议特征块中保存有 I/O 相关的函数指针供 NDIS 小端口驱动回调。
- 使用 NdisRegisterMiniport 注册时，NDIS 小端口驱动使用 **NDIS_MINIPORT_CHARACTERISTICS**（小端口特征块）数据结构携带 NDIS 小端口驱动的回调函数指针向 NDIS 框架注册。小端口特征块中也保存有 I/O 相关的函数指针供协议驱动回调。

1．协议驱动向 NDIS 框架注册

协议驱动向 NDIS 框架注册时（NdisRegisterProtocol）要执行以下几个步骤：

（1）分配一个 PROTOCOL_BINDING 数据结构，将 NDIS_PROTOCOL_CHARACTERISTICS（协议特征块）中的数据复制进去，这些数据代表了一个协议驱动。

（2）查询注册表“HKLM\SYSTEM\CurrentSet\Services\Tcpip\Parameters\Interfaces”中的设备名，这些设备名就是当前系统中的具体的网卡设备，如图 17－9 和图 17－10 所示。

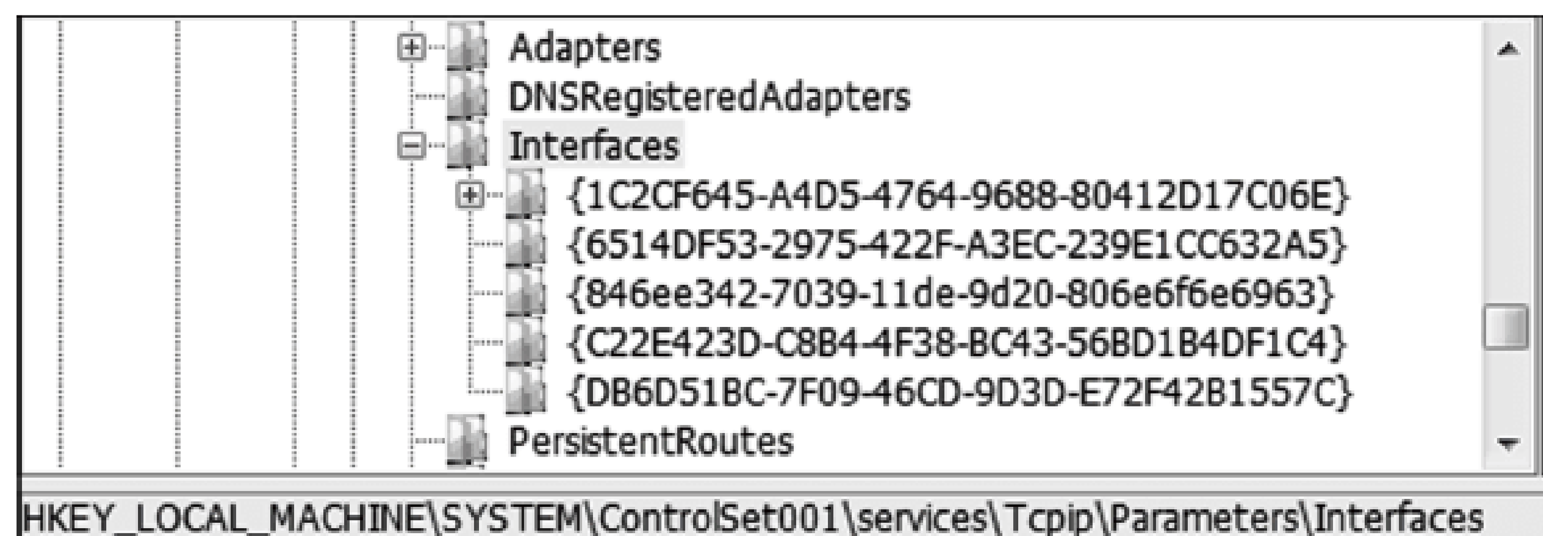


图 17－9　当前系统注册表中的网卡设备（符号链接）

{1C2CF645-A4D5-4764-9688-80412D17C06E}	SymbolicLink	\Device\NDMP5
{29898C9D-B0A4-4FEF-BDB6-57A562022CEE}	SymbolicLink	\Device\NDMP7
{3DBA63B7-82BC-467A-A3FC-CE1BA55D2BAD}	SymbolicLink	\Device\NDMP1
{4324EA8E-F737-470B-9A82-A153A9181FBA}	SymbolicLink	\Device\NDMP4
{6514DF53-2975-422F-A3EC-239E1CC632A5}	SymbolicLink	\Device\NDMP15
{71F897D7-EB7C-4D8D-89DB-AC80D9DD2270}	SymbolicLink	\Device\NDMP14
{78032B7E-4968-42D3-9F37-287EA86C0AAA}	SymbolicLink	\Device\NDMP16
{7D7C167B-2B65-48D6-B9CB-EEB11940276D}	SymbolicLink	\Device\NDMP2
{8E301A52-AFFA-4F49-B9CA-C79096A1A056}	SymbolicLink	\Device\NDMP12
{C22E423D-C8B4-4F38-BC43-56BD1B4DF1C4}	SymbolicLink	\Device\NDMP6

图 17－10　图 17－9 所示符号链接对应的设备名

（3）对上述查询到的每个设备执行协议特征块中的成员变量——绑定函数，即协议特征块的 BindAdapterHandler 指针所指向的函数，该函数是由协议驱动提供的。这一步代表了协议驱动与具体设备的绑定。

（4）将 PROTOCOL_BINDING 数据结构挂入 NDIS 的协议驱动链表 ProtocolListHead。

下面具体解释。首先查找注册表“HKLM \ SYSTEM \ CurrentSet \ Services \ Tcpip \ Parameters \ Interfaces”中的设备名，可以看到 Interfaces 目录下全是符号链接，我们通过符号链接找到了对应的真实设备名“Device \ NDMP5”“Device \ NDMP6”“Device \ NDMP15”等，而 NDMP5、NDMP6 和 NDMP15 所代表设备的类型均为 FILE_DEVICE_PHISICAL_NETCARD（如图 17－11、图 17－12 和图 17－13 所示），这是物理网卡设备对象类型，由此我们可以看出，执行协议特征块的绑定函数，其实质的被操作对象其实是物理网卡设备对象，即以物理网卡设备为参数调用协议驱动所提供的绑定函数，从而将协议驱动与每个物理网卡设备绑定关联，如图 17－14 所示。执行完 NdisRegisterProtocol 时，网卡设备对象中会保存协议驱动的各种 I/O 回调函数指针，网卡驱动可以调用协议驱动的这些回调函数。

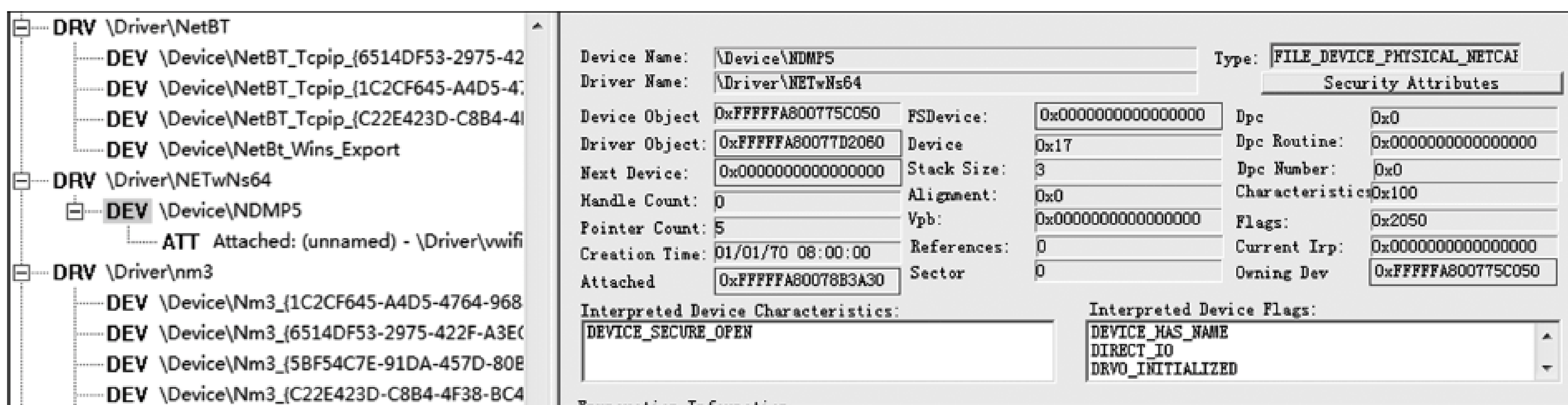


图 17－11　NDMP5 所代表的设备

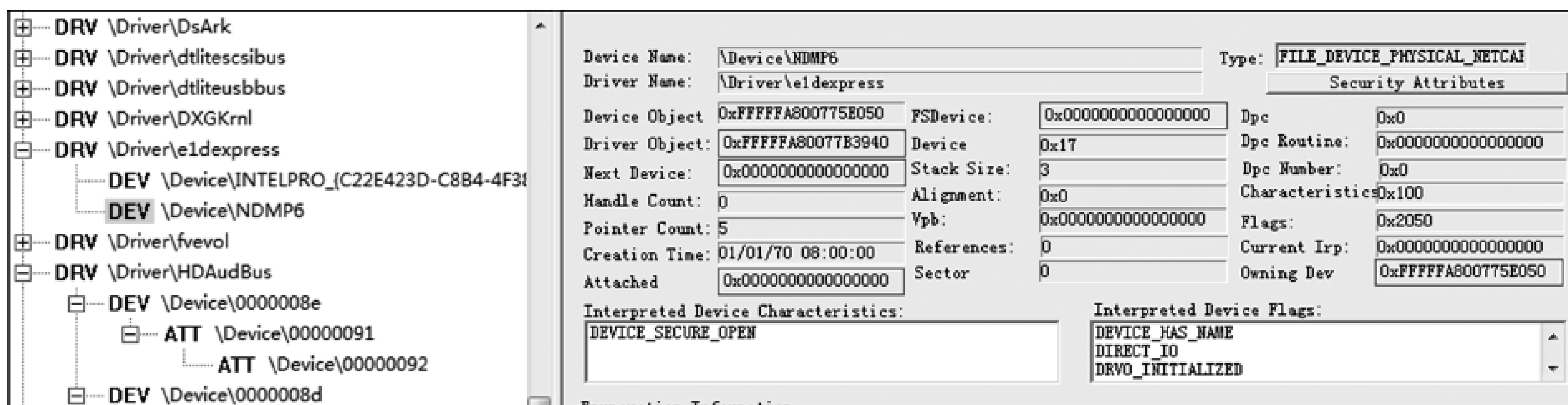


图 17－12　NDMP6 所代表的设备

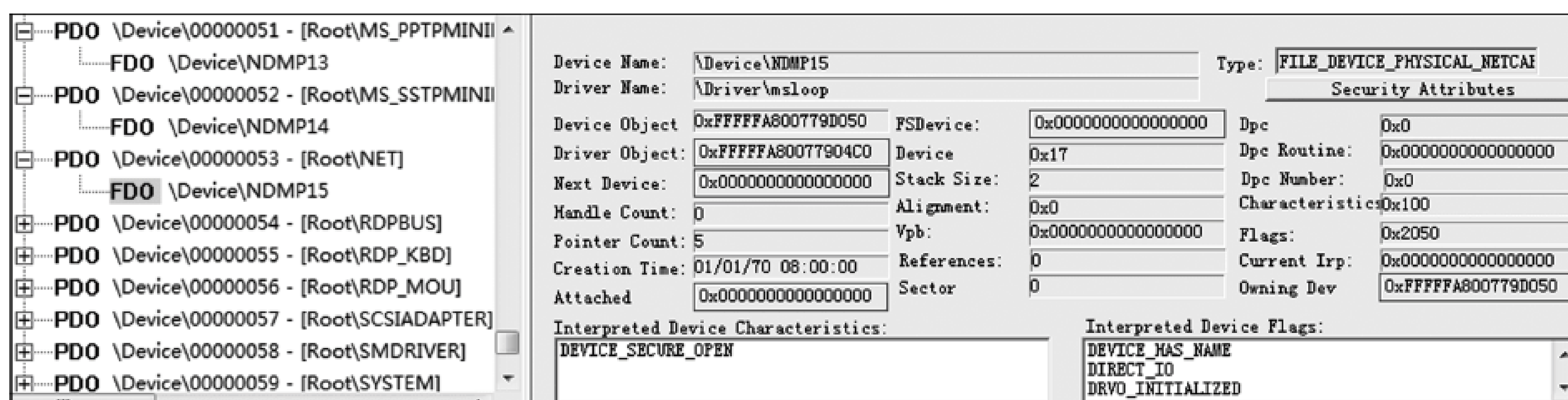


图 17－13　NDMP15 所代表的设备

从图 17－11 可以看出，NDMP5 所代表设备对应的驱动对象是 NETwNs64，其驱动载体是 netwsw04. sys（Windows 7 下），这是个无线网卡设备驱动。我们还可以看出，NDMP5 上面还堆叠了一个无名设备（类型也为 FILE_DEVICE_PHISICAL_NETCARD），其对应的驱动对象为 vwifibus，其驱动载体为 vwifibus. sys。

从图 17－12 可以看出，NDMP6 所代表设备对应的驱动对象是 eldexpress，其驱动载体是 eld62x64. sys，这是个以太网网卡设备。

从图 17－13 可以看出，NDMP15 是个功能设备对象，附加在"Device\00000053"这个物理设备对象之上。NDMP15 所代表的功能设备的类型是 FILE_DEVICE_PHISICAL_NETCARD，其驱动载体为 msloop. sys，这是个环回网卡驱动。其所附加的物理设备（下层设备）是个 FILE_DEVICE_CONTROLLER 类型的对象，驱动载体是 PnpManager，即 PNP 管理器，由此可以看出，环回网卡所对应的设备是作为虚拟设备出现的，驱动程序也是功能性驱动。从图 17－14 和图 17－15 可以看出当前系统中的各网卡设备及其与协议驱动的绑定关系。

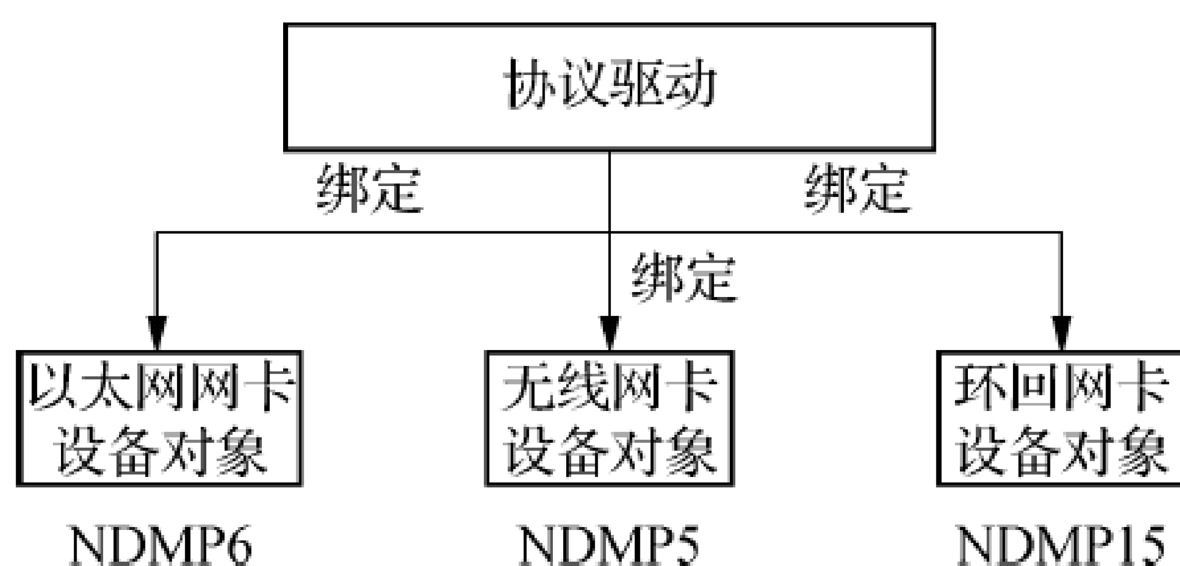


图 17－14　协议驱动与网卡设备的绑定

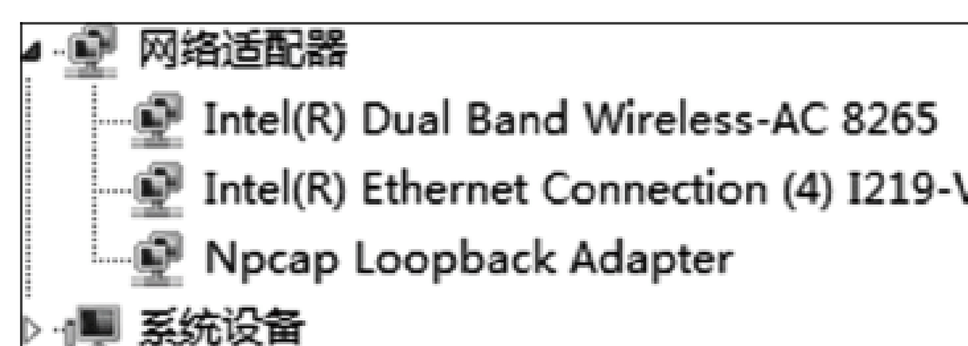


图 17－15　当前系统中出现的三块网卡设备

2. NDIS 小端口驱动向 NDIS 框架注册

NDIS 小端口驱动向 NDIS 框架注册时（NdisRegisterMiniport）则要执行以下几个步骤：

（1）将 NDIS_MINIPORT_CHARACTERISTICS（小端口特征块）中的数据复制到 NDIS 小端口驱动块（NDIS_M_DRIVER_BLOCK 数据结构）中。小端口驱动块是由网卡驱动入口函数创建的，代表了网卡设备对象。

（2）对小端口设备的驱动对象（Miniport. DriverObject）分配设备扩展区域（由 IoAllocateDriverObjectExtension 方法实现），并设置驱动对象的 PNP 处理例程和 AddDevice 函数指针。PNP 处理例程是 NDIS 提供的 NdisIDispatchPnp 方法；AddDevice 函数指针是由 NDIS 提供的

NdisIAddDevice 方法，作用是将网卡驱动创建的设备对象纳入协议栈设备对象的堆叠中。

NdisRegisterMiniport 执行完成后，系统会准备一个主功能码为 IRP_MJ_PNP、副功能码为 IRP_MN_START_DEVICE 的 IRP，并通过 IoCallDriver 方法向下传递。NdisIDispatchPnp 对这个 IRP 进行处理，它会调用由 NDIS 提供的 NdisIPnpStartDevice 函数启动网卡的运行。

3. NDIS 框架的功能函数

NDIS 框架的载体是 ndis. sys，其实也是个端口驱动模块（非小端口驱动），这个模块的入口函数也依然是 DriverEntry。但 NDIS 框架有其特殊性，它不创建设备对象，也不提供 AddDevice 函数和主功能派遣函数数组，因此不能堆叠于网络协议栈框架中，我们更多地将其看作一个运行环境或运行框架。

NDIS 提供了非常多的功能函数供框架中的成员调用（如图 17－16 所示），但实际上这些功能函数很多也是 Windows 系统调用的封装，例如 NDIS 包的分配等。因为在 NDIS 框架中，IRP 是无法通行的（PNP 类型的除外），请求的上、下行只能采用框架允许的数据结构，在这里就是以 NDIS 包代替了 IRP。

File: ndis.sys
- Dos 头部
- Nt 头部
  - 文件头部
  - 可选头部
    - 数据目录 [x]
- 段头部 [x]
- 输出目录
- 导入目录
- 资源目录
- 异常目录
- 重定位目录
- 调试目录
- Address Converter
- Dependency Walker
- 十六进制编辑
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	000C7E70	000C8D44	000C85EC	000C92C6
(nFunctions)	Dword	Word	Dword	szAnsi
0000001F	00009CE0	001E	000D260B	NdisAllocatePacket
00000020	0000B8F0	001F	000D261E	NdisAllocatePacketPool
00000021	0000B980	0020	000D2635	NdisAllocatePacketPoolEx
00000022	0000C820	0021	000D264E	NdisAllocateRWLock
00000023	00043BA0	0022	000D2661	NdisAllocateReassembledNetB...
00000024	0003F1E0	0023	000D2686	NdisAllocateSharedMemory
00000025	00012DE0	0024	000D269F	NdisAllocateSpinLock
00000026	000430E0	0025	000D26B4	NdisAllocateTimerObject
00000027	000262C0	0026	000D26CC	NdisAnsiStringToUnicodeString
00000028	0000B590	0027	000D26EA	NdisBufferLength

图 17－16 NDIS 框架提供的功能函数

NDIS 框架中存在的三个队列是在入口函数 DriverEntry 中被初始化的：

- **ProtocolListHead**：协议驱动块队列。
- **MiniportListHead**：小端口驱动块队列，代表了具体的网卡驱动模块。
- **AdapterListHead**：网络接口控制器（Network Interface Controller，NIC）即网卡设备队列。

这里还要强调一点，NDIS 不但隔离了协议驱动与 NDIS 小端口驱动，也隔离了 NDIS 小端口驱动与系统的 HAL。小端口驱动对某些芯片寄存器的读写要调用 NDIS 接口而不能直接读写，必须由 NDIS 的相关接口去访问 HAL 的接口，这样就隔离了小端口驱动与系统中代表通用芯片的硬件抽象层（HAL）。

NDIS 6.0 以上的版本也支持轻量级过滤驱动机制 NDISFilter，该机制用于截获 MAC 级别的网络数据包。NDISFilter 通过 NdisIMInitializeDeviceInstanceEx 方法创建了一个虚拟网卡，这个虚拟网卡的作用是使上层的协议驱动都绑定到它自己，并且对下层的真实网卡驱动

(NDIS 小端口驱动)解绑。NDISFilter 在网络包过滤挂钩以及多网卡绑定等场景中有应用。

17.2 NDIS 小端口驱动

1. NDIS 小端口驱动的安装与堆叠

NDIS 小端口驱动也就是物理网卡(NIC)驱动,是由各个网卡生产厂商按照 NDIS 框架规范提供的,其入口函数 DriverEntry 的主要功能包括:

- 设置小端口特征块 NDIS_MINIPORT_CHARACTERISTICS 的各个参数、函数指针等。
- 创建小端口驱动块 NDIS_M_DRIVER_BLOCK 并挂入 NDIS 框架的 MiniportListHead 队列中。
- 将小端口特征块的数据复制到小端口驱动块的对应区域内。
- 调用 NdisRegisterMiniport 函数向 NDIS 框架注册登记,这在上一节中已经介绍了。

入口函数返回以后,AddDevice 方法会被调用,这与普通的 WDM 驱动程序一样。这里的 AddDevice 方法就是 NDIS 框架的 NdisIAddDevice 函数,该函数将创建的 NIC 设备对象(FILE_DEVICE_PHISICAL_NETCARD 类型的网卡设备对象)纳入协议栈设备对象中,同时也要设置 NIC 设备的 DPC 函数,这个函数则是由小端口驱动提供的,如图 17-17 所示。

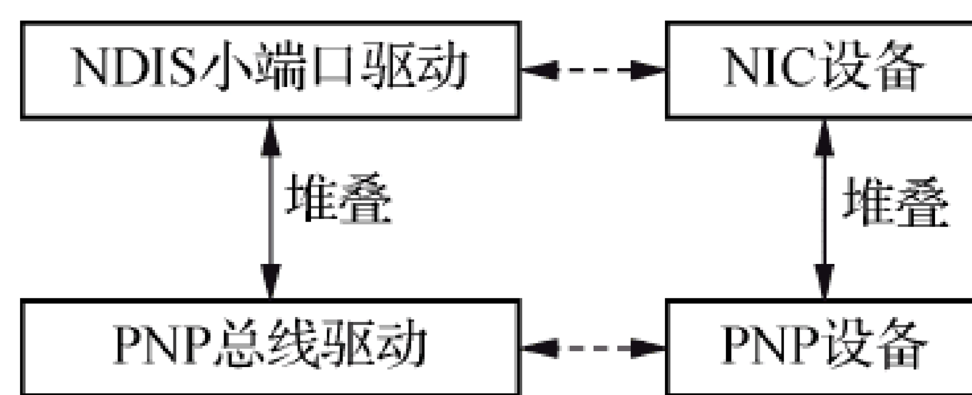


图 17-17 调用 NdisIAddDevice 后形成的设备堆叠

调用完 AddDevice 方法后,系统通过 NdisIDispatchPnp 方法下发一个主功能码为 IRP_MJ_PNP、副功能码为 IRP_MN_START_DEVICE 的 IRP 以启动网卡硬件,这也是设备启动的标准步骤。网卡的启动由 NDIS 框架提供的 NdisIPnpStartDevice 方法实现,网卡的停止也是由 NDIS 的 NdisIPnpStopDevice 实现。设备的启动与停止一般应该由设备驱动(小端口驱动)提供,但这里却由 NDIS 框架来提供,这也充分说明了 NDIS 框架的封装特性(这两个接口只是将设备驱动提供的启动和停止的功能函数包装了一下,本质上都是通过设备驱动向 NDIS 框架注册的函数),其目的就是为了让所有的网卡驱动都运行在统一的框架和接口之下,通过 NdisIPnpStartDevice/NdisIPnpStopDevice 去反调网卡驱动的启动/停止函数。

除了启动网卡,还要将 IRP 下发到下层驱动,即通过 NdisIForwardIrpAndWait 方法下发到 PNP 根驱动,其本质还是对 IoCallDriver 的封装调用。

2. 网卡的启动

我们先来看 NdisIPnpStartDevice,该方法的主要执行步骤如下:

(1) 先将 NIC 设备对象挂入 NDIS 的 AdapterListHead 队列中。NIC 设备对象的扩展区域是一个 LOGICAL_ADAPTER 数据结构,本质上是将该结构挂入 AdapterListHead 中。

(2) 调用由小端口驱动提供的初始化函数 **MiniportInitialize**。NdisIPnpStartDevice 是通

过调用小端口驱动块的 InitializeHandler 函数指针来间接调用 MiniportInitialize 的。

（3）将代表网卡设备对象的 LOGICAL_ADAPTER 数据结构也挂入小端口驱动块的设备队列。

从上述步骤可以看出，NdisIPnpStartDevice 的执行本质上还是要回调由小端口驱动（网卡驱动）提供和注册的初始化函数，因为只有具体的网卡驱动才知道怎样初始化这块网卡，NDIS 框架只不过是把调用步骤过了一下手。

由小端口驱动提供的 MiniportInitialize 的主要执行步骤如下：

（1）分配代表 NIC 设备的 NIC_ADAPTER 数据结构，并设置其中的各个属性和函数指针。

（2）从 PNP 管理器中查询资源：**MiQueryResource**。

（3）调用 NDIS 框架发布的 **NdisMSetAttributes** 方法，向 NDIS 框架报告小端口驱动所支持的 NIC 设备类型并且传递指向小端口上下文的句柄。NDIS 将在接下来的调用中把这个句柄传递给 MiniportXxx 系列函数。

（4）调用 NDIS 框架提供的 **NdisMRegisterIoPortRange** 方法，为 NdisRawReadPortXxx 和 NdisRawWritePortXxx 函数的使用建立 I/O 访问端口。

（5）调动 NDIS 框架提供的 **NdisMRegisterInterrupt** 方法，连接小端口驱动的中断服务函数（MiniportISR）和由 NIC 所产生的中断向量。其参数 NDIS_MINIPORT_INTERRUPT 也是 NIC_ADAPTER 数据结构的一部分，包含了中断服务例程 ServiceRoutine 和 DPC 例程 HandleDeferredProcessing，两者都是由 NDIS 框架提供的。当然这两者也是在函数内部反调小端口驱动提供的中断例程。

（6）调用由 NIC 驱动提供的真实设备启动函数 NICStart。

（7）将 NIC_ADAPTER 结构挂入全局数据结构 DriverInfo 的适配器队列 AdapterListHead 中。

3. 网卡的 I/O 处理

网卡的 I/O 处理过程也是一个中断处理过程，且这个过程与普通的中断处理过程并无二致。我们以网卡收到网络包为例来讲述这个处理过程，如图 17 - 18 所示。

（1）网卡收到网络包并产生硬件中断。

（2）系统分发中断，执行 IDT 中相应的中断服务例程，即中断向量的服务例程 ServiceRoutine，这个函数是在小端口驱动初始化过程中调用 NdisMRegisterInterrupt 方法所连接起来的。中断分发首先执行小端口驱动提供的中断服务例程 **MiniportISR**，再执行 **KeInsertQueueDpc** 以插入 NDIS 框架提供的 DPC 函数 **HandleDeferredProcessing**，这两者分别代表了中断的前半段与后半段。

前者（中断服务例程）的处理逻辑非常简单，只是屏蔽一下中断并插入一个 DPC，这是为了尽量缩短关中断的时间，把具体的繁杂且耗时的工作留给后者（DPC 函数）去完成，这是中断服务例程的前半段。

后半段的 DPC 函数是 NDIS 框架提供的 HandleDeferredProcessing，从中断服务例程

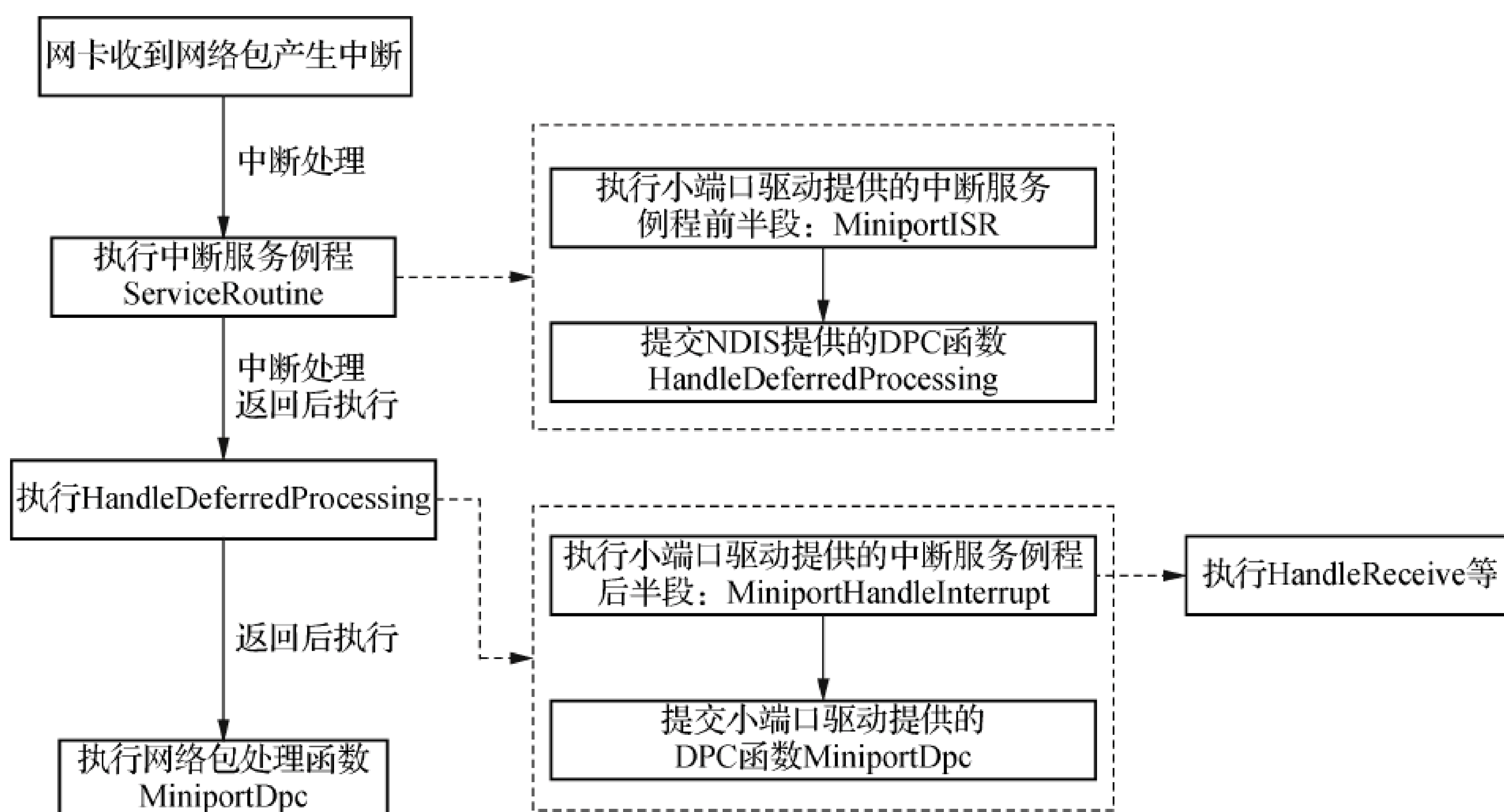


图 17－18　网卡收到数据包时的中断处理过程

ServiceRoutine 返回后就开始执行 HandleDeferredProcessing，这是因为 ServiceRoutine 中已经插入了这个 DPC 函数，执行 DPC 的过程是开中断的。

这里要强调的是，网络包在网卡驱动中的收发是一体的，完成了网络包的接收也要着手安排网络包的发送工作，所谓网络包的处理应该包含收和发两个方面。

HandleDeferredProcessing 并不能直接完成网络包的收发处理，至少不能同时完成，而是将发送的工作通过嵌套 DPC 机制来实现。HandleDeferredProcessing 本身也为网络包发送构造 DPC 函数，真正的包发送是这个嵌套插入的 DPC 函数实现的。

因此，收到一个网络包后的处理流程大致为：

（1）中断的前半段关中断，封装 DPC1 函数，即上文的 HandleDeferredProcessing，这个 DPC1 函数是实质接收网络包的 DPC 函数；

（2）中断的后半段执行上述 DPC1 函数，将接收到的网络包向上层驱动投递，并封装实质发送网络包的 DPC2 函数；

（3）在任意上下文中执行 DPC2 函数，完成网络包的发送。

其中前两步都是在中断发生时的线程中进行的，后一步则可以在任意上下文中实现。

具体来说，HandleDeferredProcessing 首先会执行小端口驱动的中断服务例程的后半段（中断处理例程被分成了前后两段），即 **MiniportHandleInterrupt**，这是由小端口驱动提供的；然后再次通过 KeInsertQueueDpc 插入一个 DPC2 函数，这个 DPC2 函数是小端口驱动提供的 **MiniportDpc**，这才是真正处理网络包发送的函数；最后开中断，当然这个开中断的方法也是要由小端口驱动提供的，因为这是针对 NIC 芯片的开中断，只有具体设备的驱动程序才知道怎么开中断。

由此来看，虽然 HandleDeferredProcessing 是由 NDIS 框架提供的，但实质上也执行由小端口驱动提供的方法：

➢ 中断服务例程后半段函数 **MiniportHandleInterrupt**,该函数调用小端口驱动提供的 HandleReceive/HandleTransmit 方法。
➢ 采用 KeInsertQueueDpc 方法将小端口驱动提供的 DPC 函数 MiniportDPC 插入 DPC 队列中,这是第二个 DPC 函数。MiniportDPC 函数从小端口驱动的工作项队列中取出一项,根据工作项的类型来调用小端口特征块中的 SendPacketsHandlers/SendHandler 等函数,最终调用 NIC 驱动提供的 NICTransmit 等接口,它也是真正处理网络包的函数。除此之外有一些信息查询和设置的操作也会在该函数中处理,例如 MiniDoRequest 函数。不过 MiniportDPC 主要用于处理网络包的发送。
➢ 采用小端口驱动提供的方法 NICEnableInterrupt 开中断。

从图 17－19 可以看出,网络包的接收并没有在第二个 DPC 中处理,实际上 HandleReceive 的调用是在执行小端口驱动的中断处理例程的后半段,这是在 HandleDeferredProcessing 中被调用的,也就是第一个 DPC 中被调用的。

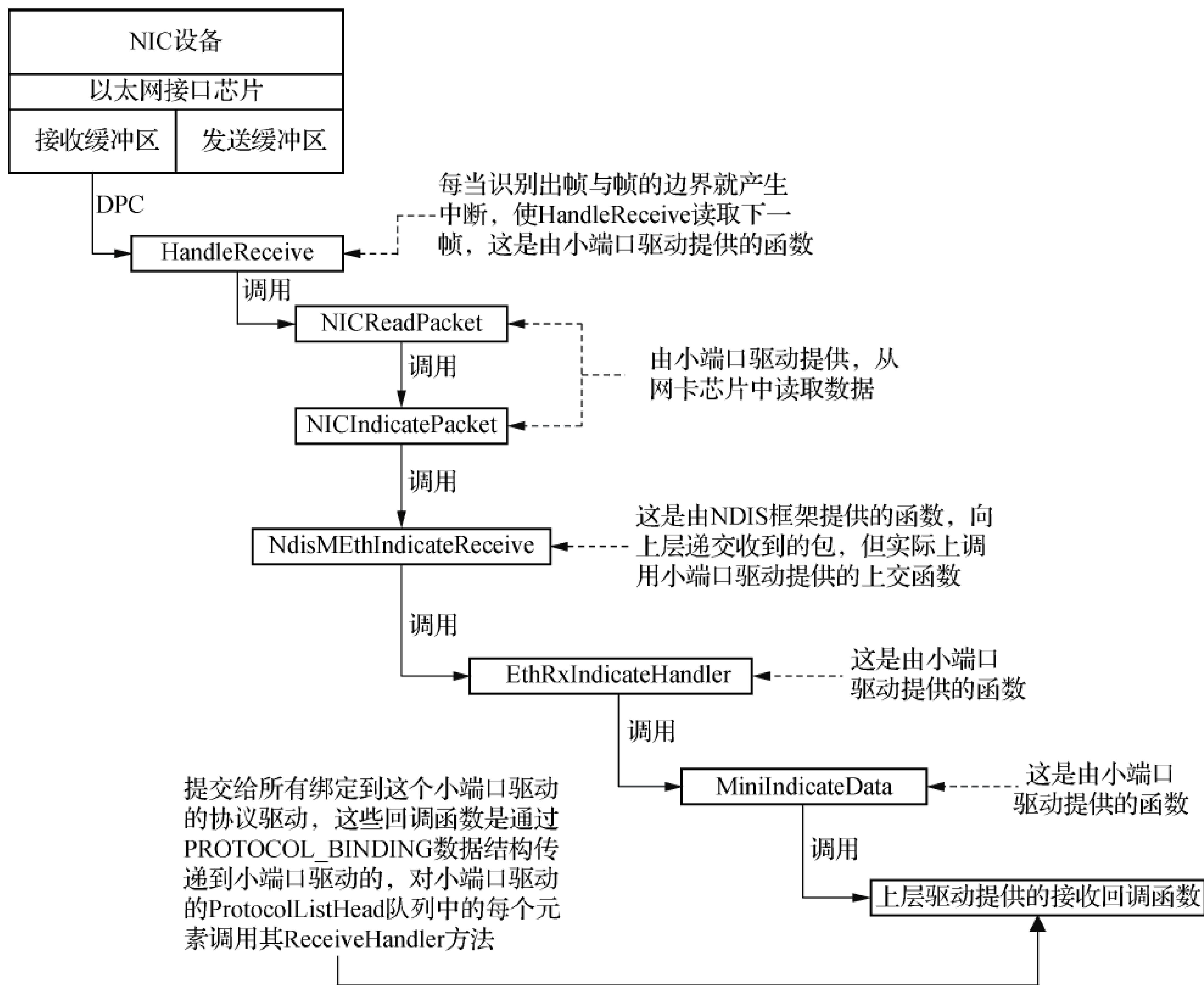


图 17－19　网卡接收到网络包后的向上投递过程

网络包的接收和发送是个复杂烦琐的过程,小端口驱动和 NDIS 框架的方法被交叉调用,这也是网络协议栈驱动的复杂性之一。

执行完上述流程后,即通过 MiniIndicateData 将网络包投递到上层驱动后,小端口驱动的中

断处理例程的后半段(MiniportHandleInterrupt)也就完成了,HandleDeferredProcessing 也随即执行完成。这里仍然是第一个 DPC。接下来执行小端口驱动提供的 DPC 函数(MiniportDpc),这个函数主要用于处理网络包的发送和信息的查询与设置,如图 17－20 所示。

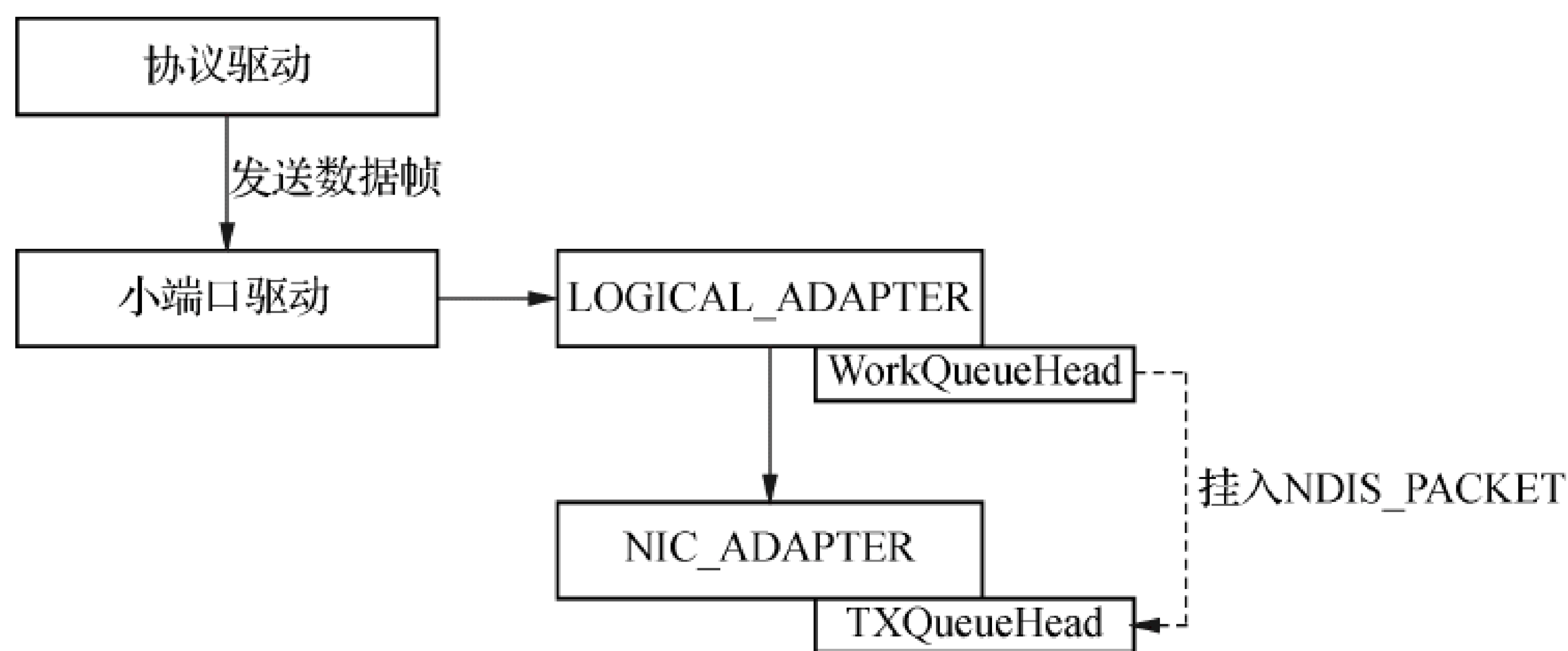


图 17－20　协议驱动向小端口驱动发送网络包的过程

在 LOGICAL_ADAPTER 结构中有个工作项队列 WorkQueueHead 专门用于发送网络包,当协议驱动向小端口驱动投递发送的网络包时首先挂入该队列,当需要发送的时候再将网络包挂入 NIC_ADAPTER 的工作项队列 TXQueueHead,这是更底层,也就是 NIC 层的发送队列,是专门与发送芯片打交道的底层队列。如果发送缓冲区为空则立刻摘下网络包发送,否则要等待 NIC 芯片发送完当前包后触发中断通知,在中断服务例程中处理 TXQueueHead 上的发送。挂载在 TXQueueHead 上的网络包是 NDIS_PACKET 结构,采用 NICTransmit 接口进行发送,从发送接口的名称中也可以感受到这个底层接口浓浓的"硬件风格"。

17.3　协议驱动

我们所说的协议驱动就是指 TCP/IP 协议驱动,具体的映像是 tcpip. sys,其大致对应协议栈的链路层、网络层和传输层的下半部分。协议驱动是 socket 机制实现的基础。如图 17－21 所示的协议驱动 tcpip. sys 的依赖库。

tcpip.sys

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntoskrnl.exe	215	001D3F88	00000000	00000000	001D3F74	0013E000
NETIO.SYS	205	001D4648	00000000	00000000	001D3F68	0013E6C0
NDIS.SYS	41	001D4CB8	00000000	00000000	001D3F5C	0013ED30
FLTMGR.SYS	2	001D4E08	00000000	00000000	001D3F50	0013EE80
fwpkclnt.sys	32	001D4E20	00000000	00000000	001D3F40	0013EE98
HAL.dll	1	001D4F28	00000000	00000000	001D3F38	0013EFA0
ksecdd.sys	20	001D4F38	00000000	00000000	001D3F2C	0013EFB0
msrpc.sys	4	001D4FE0	00000000	00000000	001D3F20	0013F058

图 17－21　tcpip. sys 的依赖库

1．协议驱动初始化

我们先来看 tcpip. sys 的入口函数 DriverEntry，在该函数中要完成如下工作：

（1）创建若干设备对象：Device\IP、Device\TCP、Device\UDP、Device\RawIP 等，这几个设备对象的类型都为 FILE_DEVICE_NETWORK，如图 17－22 所示，不过链路层是没有独立设备对象的，这几个都是传输层或网络层的设备。注意它们都属于同一个驱动对象，因此也共享同样的主功能函数。

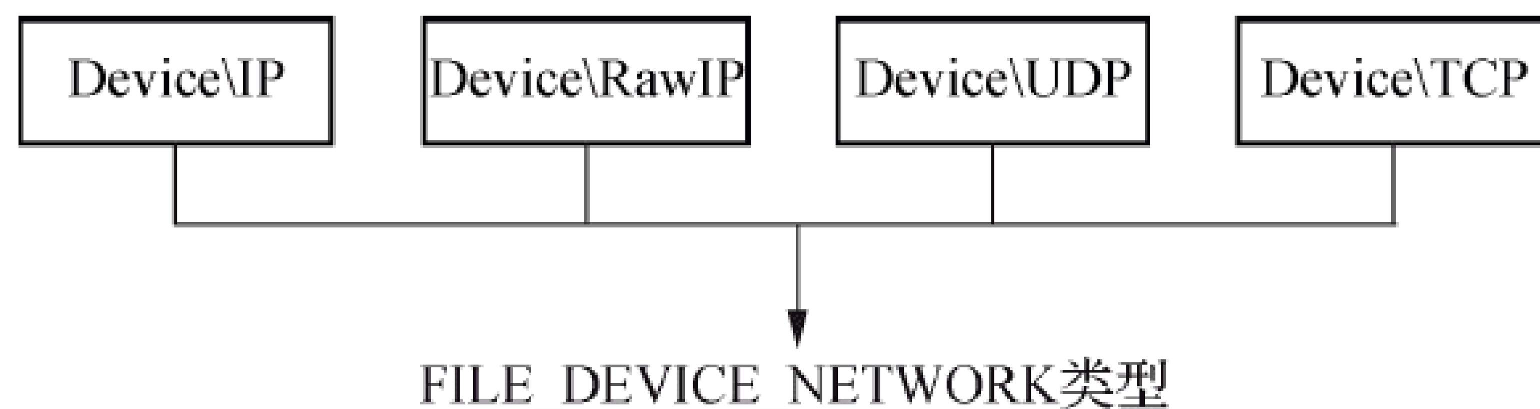


图 17－22　tcpip. sys 创建的 4 个设备对象

（2）分配用于 TDIEntityID 的数组的内存。

（3）分配用于 NDIS_PACKET 缓存数组的内存，并以全局指针 GlobalPacketPool 指向这个缓冲区。

（4）初始化三个队列：AddressFileListHead、ConnectionEndPointListHead、InterfaceListHead。

（5）针对步骤（1）中创建的 4 个设备对象执行相应的启动函数：IPStartup、RawIPStartup、UDPStartup 和 TCPStart。其中 IPStartup 的主要执行步骤包括：

① 预先分配用于 IP 数据包片段拼装恢复的缓冲区队列。

② 通过 RouterStartup 启动路由机制。

③ 使用 IPRegisterProtocol 对协议表 ProtocolTable 初始化。ProtocolTable 用于保存传输层向网络层注册的回调函数，即对应四层协议的处理函数指针，默认初始值均为 DefaultProtocolHandler。当传输层启动函数（UDPStartup/TCPStart 等）执行时再用实际的函数指针替换和覆盖。

（6）执行 LANStartup、LANRegisterProtocol、LoopRegisterAdapter。其中 LANRegisterProtocol 用于向 NDIS 框架注册协议驱动的回调函数；LoopRegisterAdapter 用于向 NDIS 框架注册环回网卡驱动的回调函数。环回网卡是个虚拟网卡，IP 包的环回流程是在 tcpip. sys 模块内部进行的，不会流转到下层驱动（小端口驱动）中。LoopRegisterAdapter 的执行步骤如下：

① 通过 ExInitializeWorkItem 方法创建一个环回工作项（LoopWorkItem），该工作项用于处理环回包的接收（实际处理函数为 LoopReceiveWorker），一旦有需要即将其挂入内核工作线程的工作项队列中。

② 初始化绑定信息的参数，包括环回网卡发送函数指针 LoopTransmit 等，并以绑定信息为参数通过 IPCreateInterface 方法为环回网卡创建一个 IP_INTERFACE 数据结构，该结构代表一块网卡。

③ 通过 IPRegisterInterface 将上述创建的 IP_INTERFACE 结构添加到路由表中，同时也将其挂入 InterfaceListHead 队列中，该队列是个全局队列，代表所有的网卡结构的集合。

(7) 设置 TCP/IP 协议驱动的主功能函数:

- **IRP_MJ_INTERNAL_DEVICE_CONTROL**:TiDispatchInternal;
- **IRP_MJ_DEVICE_CONTROL**:TiDispatch;
- **IRP_MJ_CREATE**、**IRP_MJ_CLOSE**、**IRP_MJ_CLEANUP**:TiDispatchOpenClose。

协议驱动不支持 IRP_MJ_READ 和 IRP_MJ_WRITE 两个主功能码,网络报文的收发是通过功能码 IRP_MJ_INTERNAL_DEVICE_CONTROL 实现的。但在 AFD(辅助功能驱动)中,IRP_MJ_READ 和 IRP_MJ_WRITE 是起作用的,NtReadFile 和 NtWriteFile 正是通过这两个主功能码实现了报文的读写。

(8) 初始化网络事件超时处理函数(IPTimeout),并使之作为一个工作线程的工作项;初始化网络事件超时的 DPC 机制(IPTimeoutDpcFn);初始化网络事件定时器 IPTimer。

tcpip. sys 没有 AddDevice 函数,这意味着上述 4 个设备对象不会被堆叠到别的设备之上,至少不会通过 AddDevice 来堆叠设备。因此执行完入口函数 DriverEntry,tcpip. sys 就可以正常工作了。接下来我们来探讨一下 tcpip. sys 模块的数据接收和发送。

2. 协议驱动接收网络包

在小端口驱动中,网卡接收到网络包(数据帧)之后一路回调,最后对小端口驱动的 ProtocolListHead 队列中的所有元素(每个元素都代表了一个协议驱动)调用 ReceiveHandler 方法,在 tcpip. sys 模块中这个方法指向 ProtocolReceive 函数。我们来看这个函数的执行过程:

(1) 通过 AllocatePacketWithBuffer 分配和初始化 NDIS 数据包 NdisPacket。

(2) 通过 NdisTransferData 方法将网卡驱动回调上来的数据帧转化为 NdisPacket。NdisTransferData 虽然是由 NDIS 框架提供的,但鉴于各个网卡驱动回调上来的网络帧可能格式不一样,因此 NdisTransferData 会通过调用各个网卡驱动的 TransferDataHandler 函数来复制帧,只有各个网卡驱动才知道这些网络帧要怎么转化。

(3) 执行 ProtocolTransferDataComplete,将 NdisPacket 上交到网络层。由于链路层设备没有独立的设备对象,因此没有办法被上层应用直接调用,只能在这里继续向上提交。ProtocolTransferDataComplete 将 LanReceiveWorker 函数作为工作项插入内核工作线程中,且这个工作项是被插入紧急队列 CriticalWorkQueue 中的。

在不同的中断请求级别下 LanReceiveWorker 的执行时机是不一样的,如图 17-23 所示首先判断是最低优先级还是更高优先级:

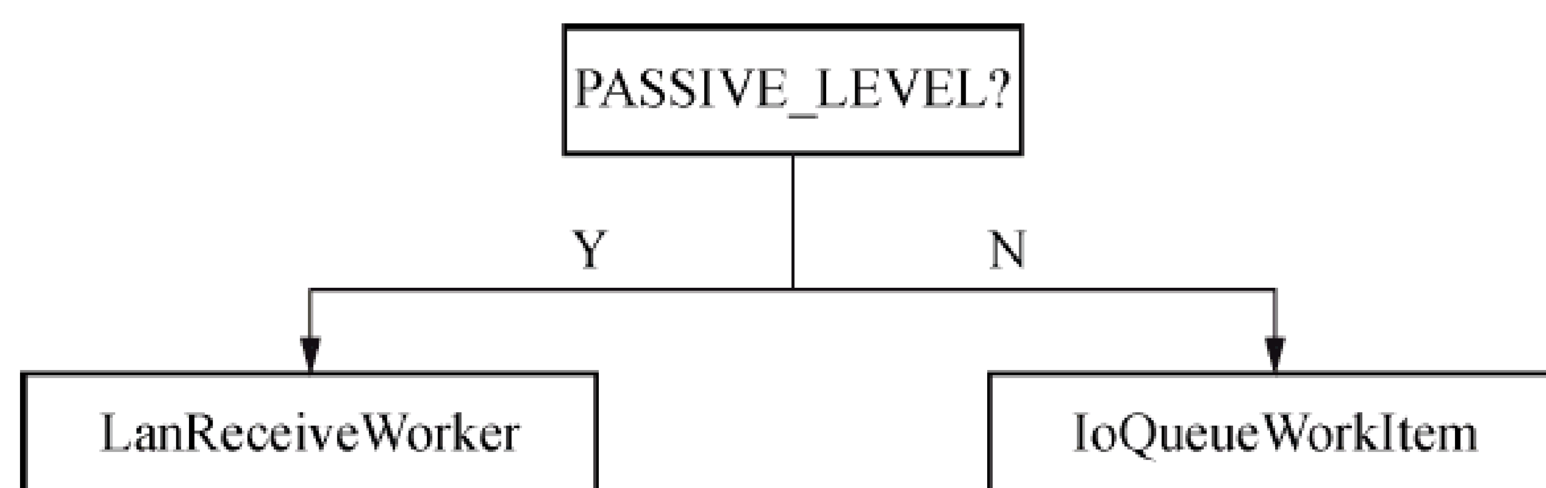


图 17-23 不同中断请求级别下 LanReceiveWorker 的执行时机

- 在 PASSIVE_LEVEL 级别时直接执行;

➢ 在更高级别时需要等降到 PASSIVE_LEVEL 级别时才可以执行，因此将该工作项挂入内核的工作队列 CriticalWorkQueue 是为了等待中断请求级别下降，这就是 IoQueueWorkItem 的作用。当内核工作线程被调度时这些工作请求自然会被逐一执行。

LanReceiveWorker 的执行流程如图 17－24 所示：

（1）通过 NdisGetFirstBufferFromPacket 从 NDIS 包中获取所承载的以太帧。

（2）获取以太帧后调用 IPReceive/ARPReceive 函数将数据向上层回调。

（3）最后通过协议表 ProtocolTable 对应的索引（TCP/UDP 等）获取对应的四层处理函数（TCPReceive/UDPReceive）进行处理，这时网络包上升到了传输层。

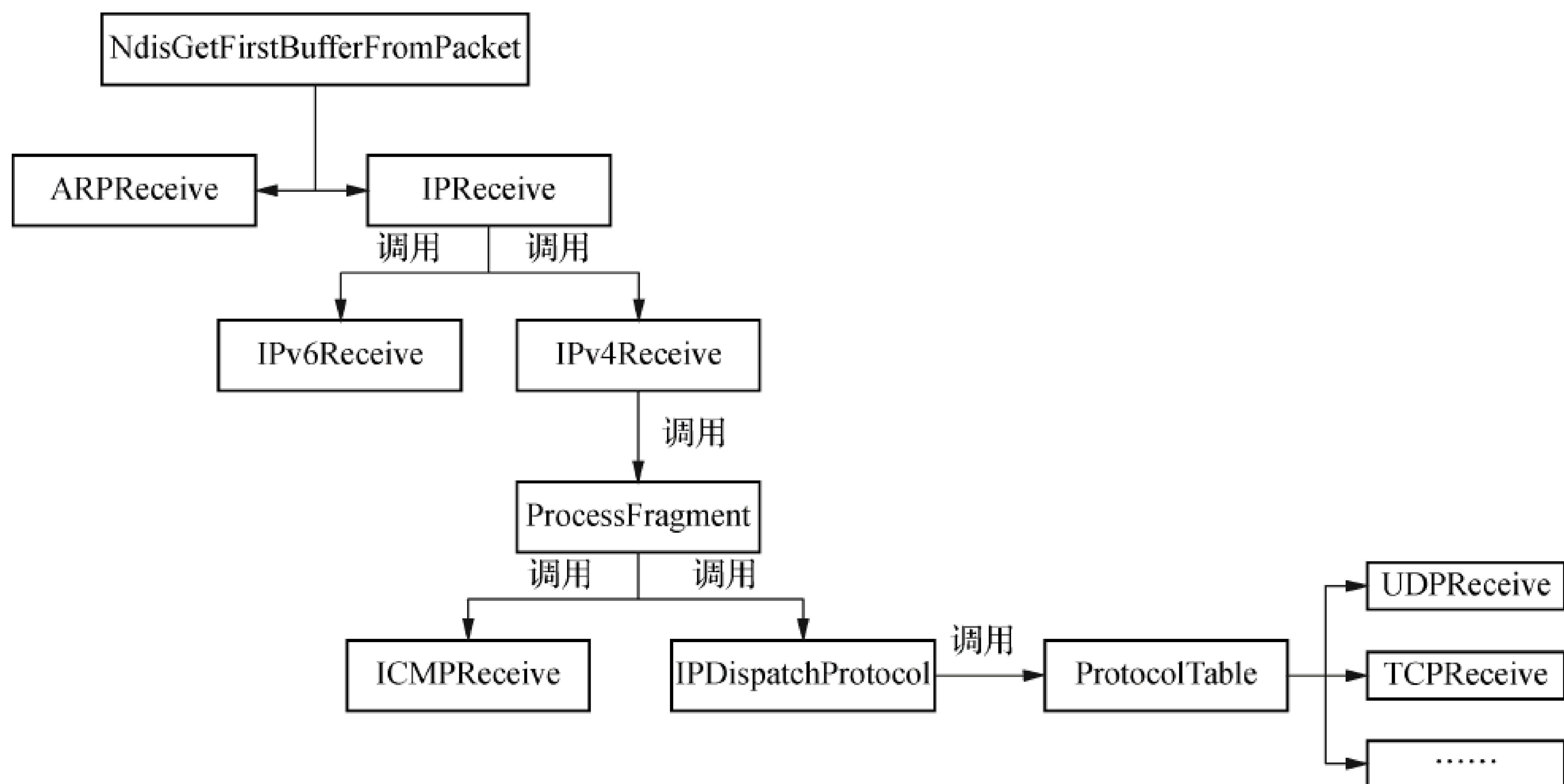


图 17－24　LanReceiveWorker 的执行流程

进入四层协议后，我们以 UDPReceive 为例来讲解在传输层的回调：

（1）通过 AddrSearchFirst，以目标 IP 地址和目标端口（Port）为索引在路由表中搜索地址文件，地址文件描述了协议类型以及目的端的 IP 地址和端口。

（2）如果发现了相符合的插口（socket）则调用 DGDeliverData 方法投递这个网络包。

（3）如果没有发现相符合的 socket，则向下回复 ICMP 消息，表示端口或地址不可达。

在路由表 AddressFileListHead 队列中存放了若干地址文件结构 ADDRESS_FILE，每个这样的结构代表了一个插口，就是我们常说的 socket。在 socket 中包括了三元组数据：协议类型（TCP/UDP）、目的端 IP 地址、目的端端口。无论是 TCP 还是 UDP，都存在上述三元组数据，以便于相同协议的数据包寻找目的地址。

进行到这里，数据包在传输层的“游历”就告一段落了，再往上就是通过 socket 向应用程序返回收到的数据了。这就是 AFD 要做的事情了。

3．协议驱动发送网络包

仍以 UDP 为例，在 tcpip.sys 中 UDP 报文的发送采用的是 IPSendDatagram 方法（本质上是调用 SendFragments 方法）：





（1）通过 AllocatePacketWithBuffer 创建和分配一个 NDIS 包 NdisPacket。

（2）调用 IPSendFragment 发送上述 NdisPacket，这个 NdisPacket 是发往相邻节点的。

所谓相邻节点就是 NEIGHBOR_CACHE_ENTRY（NCE）数据结构，这个数据结构包含了目的节点的 MAC 地址等信息。我们知道，网络包发送和接收的时候，在网关或目的端之间都是以 MAC 地址为标识进行传输的。例如在局域网中，NEIGHBOR_CACHE_ENTRY 的 MAC 地址可能是目的端的 MAC 地址；而在跨网出局域网传输的情况下，MAC 地址可能就是网关的 MAC 地址了。IPSendFragment 就是将 NDIS 包发送到这个 MAC 地址，发送完了就不管了，就认为已经发送到目的端了（这也是 UDP 的特点），不同层的协议和驱动不会单独考虑最终的发送结果，发送的结果应该由整个协议栈和对应的驱动一起保证。

IPSendFragment 的执行步骤还是比较繁琐的：

（1）分配一个 NEIGHBOR_PACKET 数据结构，使用 NdisPacket 初始化该结构。

（2）将 NEIGHBOR_PACKET 结构挂入 NCE 的 PacketQueue 队列。

（3）通过 NBSendPackets 启动队列发送机制：先从 NCE 的 PacketQueue 队列中取出一项，再调用 NCE 的 Interface（代表网卡）的 Transmit 方法进行发送，这是个链路层的发送函数，在 tcpip. sys 中的实现是 LANTransmit。

稍早一点版本的协议驱动不仅仅实现了常规的 TCP/IP 协议，也可以通过添加过滤驱动或采用挂钩机制实现 IPSec、IP 过滤器、防火墙和 NAT 功能，如图 17－25 所示。不过随着 Windows 版本的演进，新的网络体系结构（Window 过滤平台）将这些具体的网络应用功能剥离了出来。

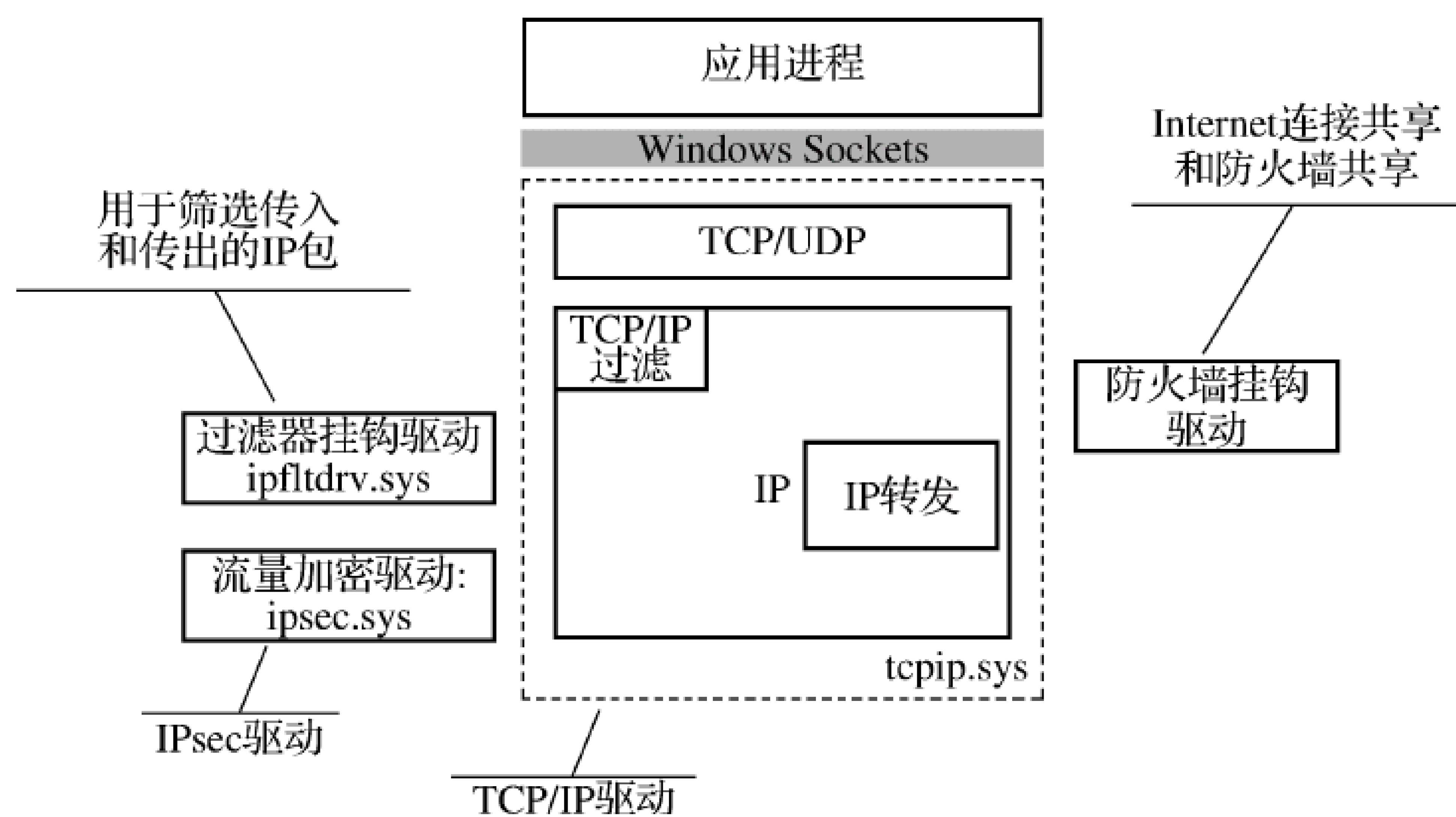


图 17－25　tcpip. sys 中用于 IP 包处理的组件

17.4　AFD

AFD（辅助功能驱动）是 Windows socket 机制的重要组成部分。一般来说，socket 机制的实现可分为用户态空间和内核态空间两部分，用户态空间的实现是 ws2_32. dll 和 mswsock.

dll 等(ws2_32. dll 只是公布 socket 接口, mswsock. dll 才是具体 socket 功能的提供者), 而内核态空间的实现就是 AFD。

AFD 也是以驱动的方式存在于 Windows 中的, 其映像文件是 afd. sys。图 17 – 26 说明了 AFD 所依赖的其他的系统或 DLL 模块。

afd.sys

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntoskrnl.exe	200	00077A38	00000000	00000000	00077A24	00015000
TDI.SYS	6	00078080	00000000	00000000	00077A1C	00015648
NETIO.SYS	28	000780B8	00000000	00000000	00077A10	00015680
msrpc.sys	10	000781A0	00000000	00000000	00077A04	00015768

图 17 – 26　afd. sys 的依赖库

可以看到, afd. sys 除了依赖内核(ntoskrnl. exe)、NetBIOS 等, 还依赖我们在后面章节要探讨的 TDI 模块。AFD 的入口函数仍然是 DirverEntry, 其主要工作是创建 AFD 设备对象, 初始化功能函数数组。在 AFD 中, IRP_MJ_READ 和 IRP_MJ_WRITE 主功能码是有效的, 与 IRP_MJ_CREATE、IRP_MJ_CLOSE 和 IRP_MJ_DEVICE_CONTROL 一样, 对应的主功能函数都是 AfdDispatch, 在 AfdDispatch 方法中根据副功能码的不同再区别对待, 如图 17 – 27 所示。



图 17 – 27　AFD 主功能函数的子处理函数





AFD 依然没有提供 AddDevice 方法，因此 AFD 设备对象不会堆叠在其他设备之上，这些设备就是 TCP/IP 协议驱动所形成的设备（Device \ UDP、Device \ TCP 等）。但是 AFD 中 socket 需要向下传输的数据并不会因为设备没有堆叠就不往下传了，而会通过 TDI 向下传输。由于 tcpip. sys 内部不能使用 IRP 传递读写数据，因此堆叠了也没什么用，IRP 传不下去，因此要在 TDI 层将 IRP 转化为 NDIS 包才能继续向下传递。

虽然 AFD 设备没有堆叠在其他设备之上，但是其他驱动生成的设备却可以堆叠在 AFD 设备之上以便过滤 socket 信息，如图 17－28 所示，360 杀毒软件生成的 AntiHacker 设备就堆叠在 AFD 设备之上。

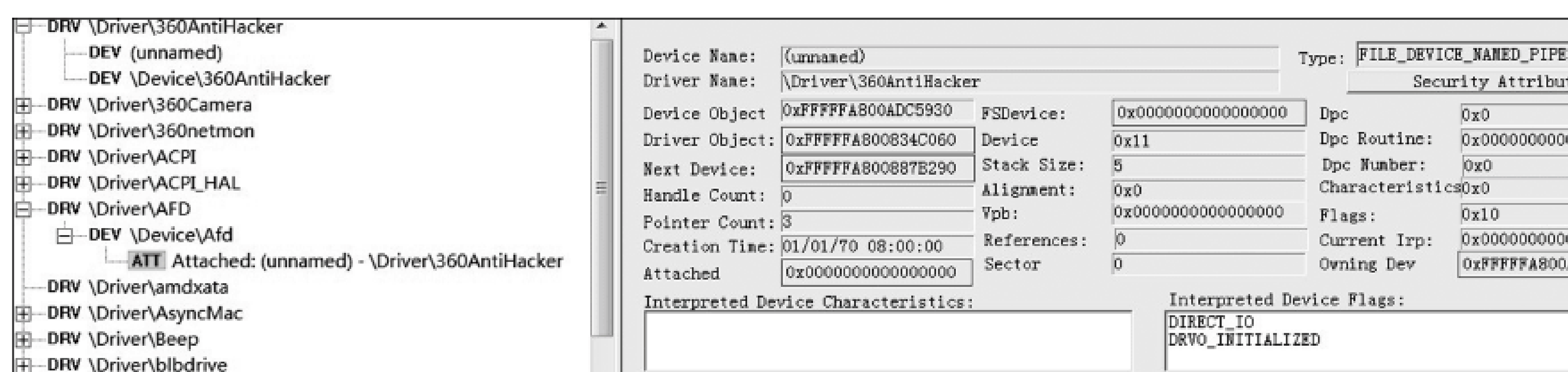


图 17－28　AFD 设备对象及其之上的设备

AFD 在系统中的位置如图 17－29 所示，其实现了传输层的偏上层的逻辑以及 socket 的基础操作。socket 机制在用户态空间的实现库 ws2_32. dll 和 mswsock. dll 等模块调用 ntdll. dll 中的相关系统调用，这些系统调用通过 I/O 管理器向 AFD 发送对应的 IRP，这些 IRP 在 AFD 中被翻译成 NDIS 包传递到 tcpip. sys 模块中。由于 Windows 不直接提供 socket 操作 API，只提供文件操作 API，因此必须有个“中介机构”将 socket 操作映射到文件操作 API 上，这就是 ws2_32. dll 和 mswsock. dll 等模块的由来（mswsock. dll 是一个传输服务的实现者，通过辅助库与内核驱动打交道，并且实现了一些 WinSock 的扩展函数；wshtcpip. dll 则是 TCP/IP 协议的辅助库）。

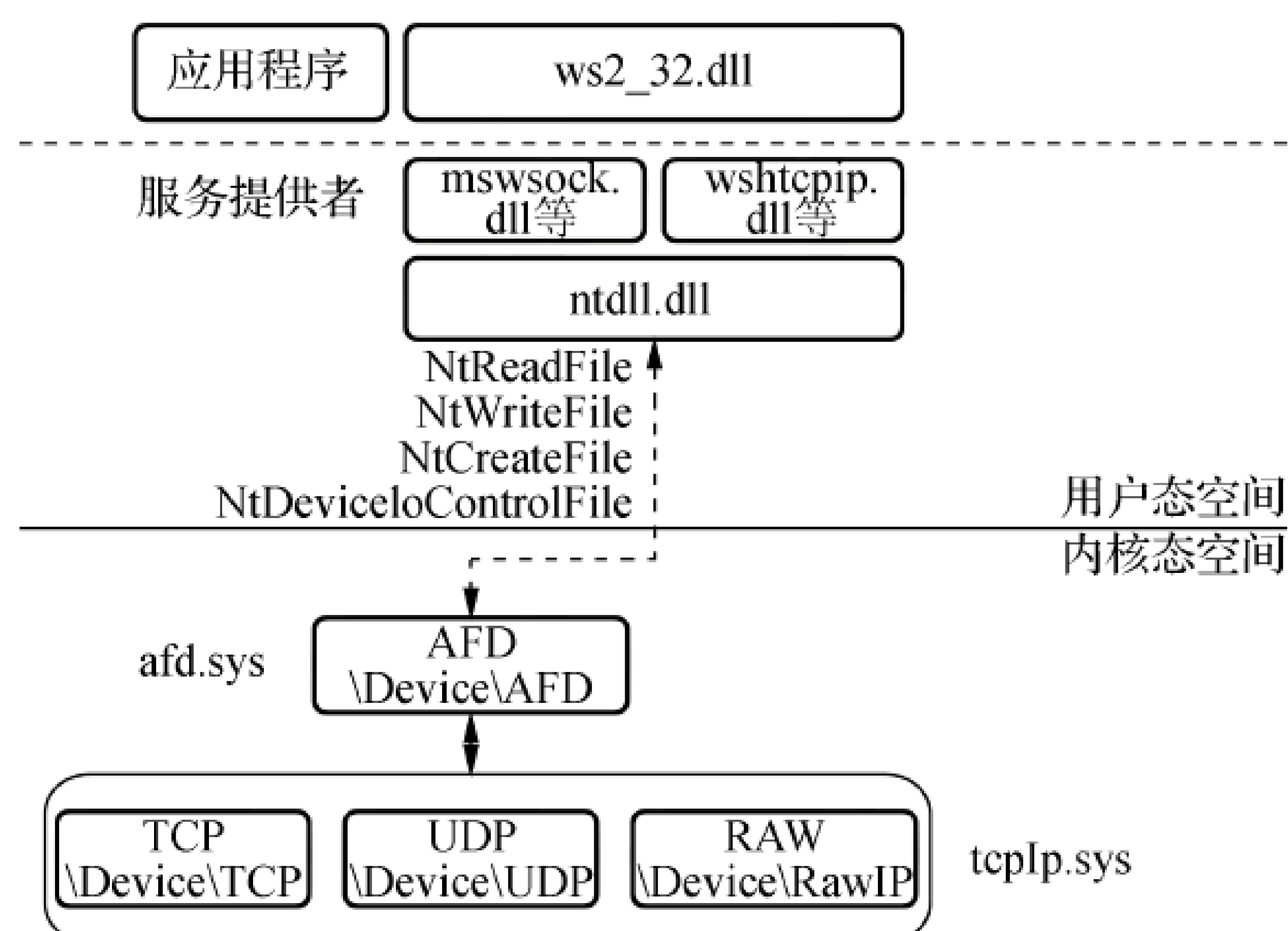


图 17－29　AFD 在系统中的位置

在向下传递的过程中，TDI 起着隔离 AFD 与协议驱动的作用。TDI 定义了主功能码和

副功能码的含义，即使下层的协议驱动不是 tcpip. sys，只要遵循 TDI 规定的功能码定义也可以自己实现 TCP/IP 协议的功能。

1. 创建与绑定 socket

下面我们来看一下 socket 的创建与绑定。

socket 的创建是通过 ws2_32. dll 中的 WSPSocket 方法实现的，WSPSocket 的工作流程大致如下：

（1）通过 SockGetTdiName 根据地址簇、socket 类型、协议类型确定 TDI 的服务提供者的设备名。其中：

① 地址簇一般选择 AF_INET\AF_INET6 类型，表示在 Windows 环境下的 IPv4\IPv6 协议。

② 常用的 socket 类型包括 SOCK_STREAM（流式传输）、SOCK_DGRAM（数据包式传输）、SOCK_RAW 等。

③ 协议类型包括 TCP、UDP、ICMP 等，分别对应 socket 的上述几种类型。因此，确定的 TDI 服务提供者的设备名包括 Device\TCP、Device\UDP 和 Device\RawIP，这也正是在 tcpip. sys 中创建的几个设备对象。

（2）根据上一步获得的信息创建并初始化 socket。

① socket 在内核中的设备对象是 Device\Afd\Endpoint，每次创建 socket 时，都要打开这个设备对象（第一次创建 socket 时是创建 Endpoint，创建后也要打开）。

② 既然是打开，就会有相应的文件对象代表本次的打开会话，这是因为每次打开时的参数可能不同，因此无法在一个设备对象中保存所有的打开会话及其参数，只能是每次打开都由一个数据结构保存这些信息，这就是文件对象的由来。

③ 打开 socket 的文件对象 AFD_FCB，这是专门用来存储 socket 信息的数据结构。打开 socket 的时候，通过系统调用 NtCreateFile 将 socket 的各种参数作为扩展信息传入并保存在 AFD_FCB 中。

④ 这里要注意，Device\Afd\Endpoint 设备对象名中的“Device\Afd”，这是个在第一次打开 socket 时就应该已经存在的设备对象，因此创建/打开 socket 时，文件系统解析到这里会首先执行 Device\Afd 的设备解析函数 IopParseDevice。IopParseDevice 的主要功能包括：

a）判断“Device\Afd\Endpoint”是否存在，如果不存在就创建并打开，否则就只是打开，打开意味着创建一个文件对象 AFD_FCB 来代表本次对 socket 的访问，打开的步骤见下一步。

b）下发主功能码为 IRP_MJ_CREATE 的 IRP 给 AFD，表示创建 socket。AFD 调用主功能函数 AfdDispatch，AfdDispatch 再调用 AfdCreateSocket 来创建 socket。

AfdCreateSocket 的核心操作就是创建一个 AFD_FCB 数据结构，并利用 NtCreateFile 中传下来的扩展信息初始化 AFD_FCB。AFD_FCB 中比较重要的数据结构有：

➢ **PendingIRPList**：这是一个 IRP 队列数组，用于暂存异步操作的 IRP。

- 许多网络操作是可以并行的，因此根据操作类型分别暂存 IRP 有助于加快并行进度。
- 这些 IRP 共分为 6 类，包括 CONNECT、RECV、SEND、PREACCEPT、ACCEPT 和 CLOSE，与 socket 的操作基本一致。

➢ **DatagramList**：协议驱动已经接收到，但应用进程尚未取走的 UDP 包。

➢ **PendingConnections**：协议驱动已经接收到，但应用进程尚未取走的 TCP 包。

➢ **Recv. Window**：接收缓冲区大小。

➢ **Send. Window**：发送缓冲区大小。

因此，执行完 AfdCreateSocket，socket 的创建就算完成了。下面我们再来看看 socket 的绑定。

所谓绑定就是将 socket 与 IP 地址和端口相关联。最初创建的 socket 只是个空壳，要使用 socket 则必须将它与某个实际地址相关联，使它代表这个地址。socket 绑定是通过 ws2_32. dll 中的 WSPBind 方法实现的，WSPBind 又通过系统调用 NtIoControl 将绑定请求下发到 AFD 中。

socket 的绑定操作的主功能码为 IRP_MJ_DEVICE_CONTROL，副功能码为 IOCTL_AFD_BIND，在 AfdDispatch 方法中其对应的处理函数为 AfdBindSocket。

AfdBindSocket 的核心操作包括：

（1）执行 WormSocketForBind。调用 tdi. sys 模块中的 TdiOpenAddressFile 方法，通过这个方法调用 TdiOpenDevice 以打开 Device\UDP 或 Device\TCP 等设备对象，而 TdiOpenDevice 是调用 ZwCreateFile 完成这个打开操作的。

当 ZwCreateFile 将请求转换成 IRP 并发送给 tcpip. sys 的时候，协议驱动用于处理这个操作的主功能函数是 TiDispatchOpenClose：

➢ 对于 IRP_MJ_CREATE 主功能码操作，TiDispatchOpenClose 调用 TiCreateFileObject；

➢ 对于 IRP_MJ_CLEANUP 主功能码操作，TiDispatchOpenClose 调用 TiCleanupFileObject。

以 TiCreateFileObject 的 UDP 方式绑定为例，绑定 socket 时调用的是 FileOpenAddress 方法（建立连接时调用 FileOpenConnection 方法，其他操作时调用 FileOpenControlChannel 方法）。FileOpenAddress 首先分配一个地址文件 ADDRESS_FILE 结构，地址文件代表了一个 IP + PORT；然后对 ADDRESS_FILE 进行初始化，ADDRESS_FILE 的 Send 函数指针设置为 UDPSendDatagram，并初始化 ReceiveQueue 和 TransmitQueue 两个队列；最后将 ADDRESS_FILE 结构挂入 AddressFileListHead 队列中，如图 17 – 30 所示。至此 TiCreateFileObject 执行完毕。

（2）执行 TdiReceiveDatagram。启动下层的接收机制，即挂入一个接收使能请求以表达接收的意愿。这个接收请求是一个类型为 TDI_RECEIVE_DATAGRAM 的 IRP，并且该 IRP 的完成例程为 PacketSocketRecvComplete。

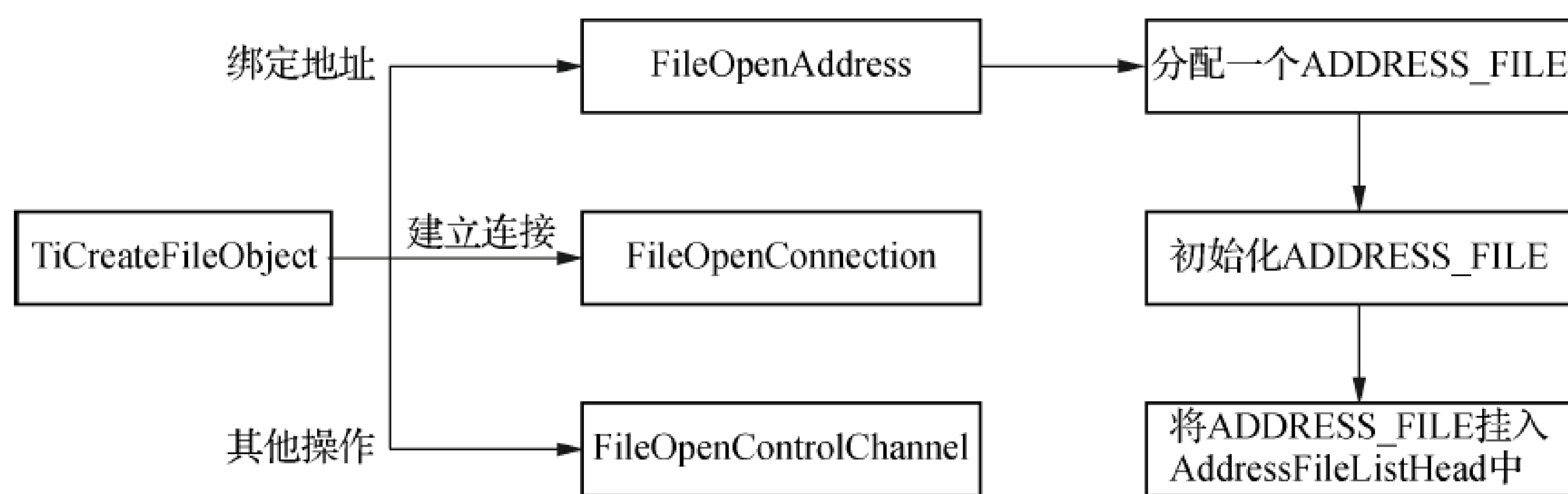


图 17 - 30　TiCreateFileObject 的调用流程

tcpip. sys 处理这个 IRP，将这个请求挂入地址文件 AddressFile 的 ReceiveQueue 队列中，并设置该请求的完成函数为 DGReceiveComplete，用户完成函数为 DispDataRequestComplete。其具体的执行过程我们在"UDP 包的接收"部分会详细介绍。

2. UDP 包的发送

为了加深理解，我们再来考察一下 UDP 包的发送和接收，先来看 UDP 的发送。

AFD 的 AfdPacketSocketWriteData 函数是发送 UDP 包的具体执行者，其调用 TdiSendDatagram 方法进行以下操作（如图 17 - 31 所示）：

（1）创建主功能码为 IRP_MJ_DEVICE_CONTROL、副功能码为 TDI_SEND_DATAGRAM 的 IRP，这是通过 TdiBuildInternalDeviceControlIrp 方法实现的。

（2）使用 TdiBuildSendDatagram 进一步构造 IRP，这一步主要是将某些函数指针赋值给 IRP。

（3）调用 TdiCall 方法下传 IRP。注意，这里 TdiCall 的参数 DeviceObject 是 tcpip. sys 创建的 Device\UDP 设备对象。TdiCall 本质上是调用 IoCallDriver 进行 IRP 的下传，并等待下传事件的完成。

（4）tcpip. sys 收到 IRP 后调用功能函数 TiDispatchInternal，当鉴别出副功能码为 TDI_SEND_DATAGRAM 时调用子功能函数 DispTdiSendDatagram 进行发送。

（5）DispTdiSendDatagram 搜索地址文件，找到符合源地址的 AddressFile（ADDRESS_FILE 类型），并调用地址文件的 Send 方法，在这里就是 UDPSendDatagram。

（6）UDPSendDatagram 首先创建一个 UDPPacket（由 BuildUDPPacket 创建），再调用 IPSendDatagram 方法进行发送。IPSendDatagram 发送时不是通过 IP 地址和端口号来发送，而是通过 NCE（NEIGHBOR_CACHE_ENTRY）来发送，这表明 IPSendDatagram 的处理已经到了链路层。

上述步骤中，IPSendDatagram 在发送的时候会将完成函数指针传进去，也就是说发送完成后完成函数 UDPSendPacketComplete 会被调用，这是 tcpip. sys 中的函数，用于通知 tcpip. sys 模块 UDP 包已经发送完成。

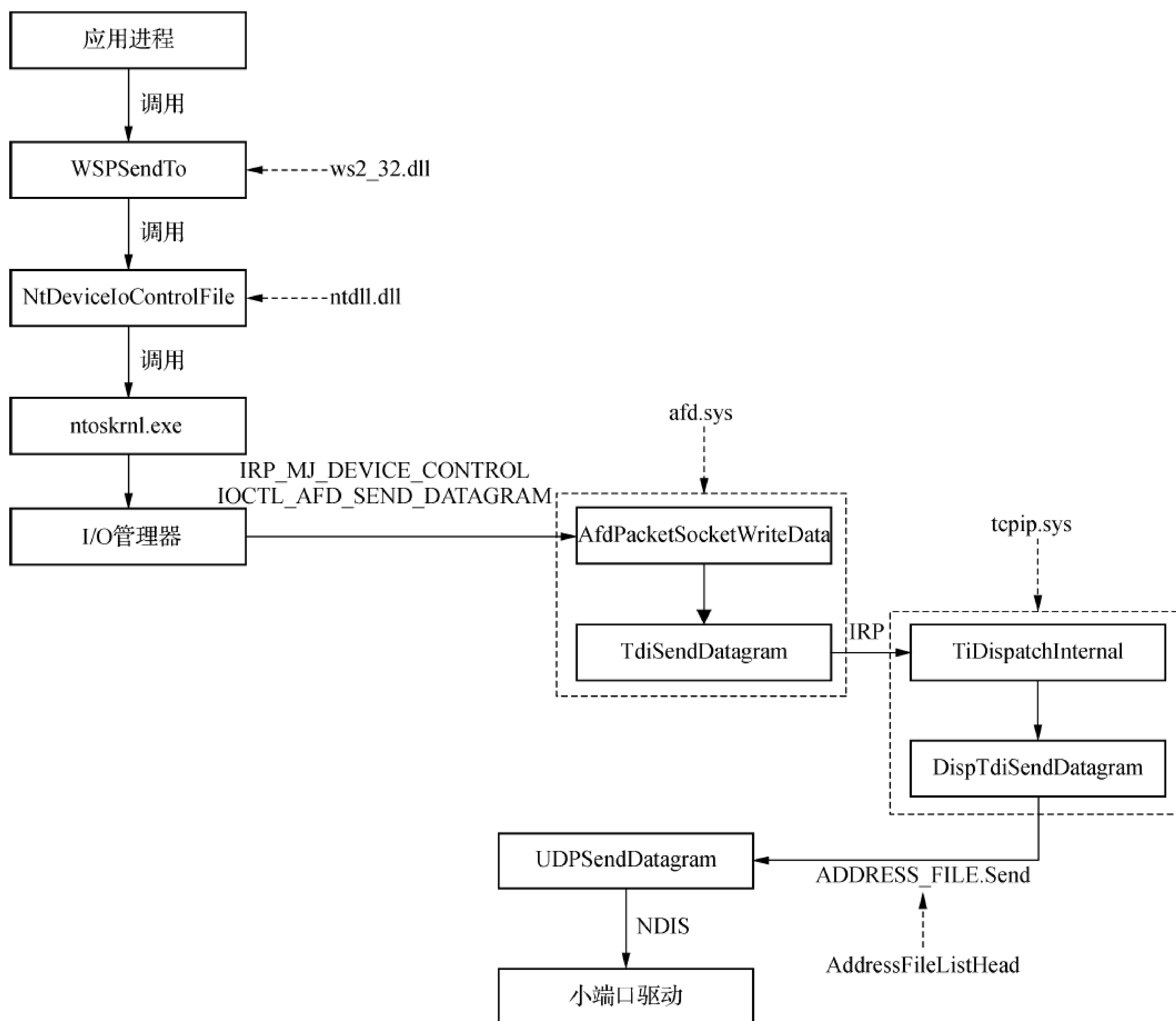


图 17－31　UDP 包发送的全流程

从图 17－31 的流程来看，创建和绑定 socket 时也下发了一个表达接收意愿的请求，这相当于给出了一个初始的动能。

3. UDP 包的接收

接下来我们来考察较为复杂的 UDP 包的接收。

与发送类似，接收也是从应用进程、ws2_32、I/O 管理器这条纵线开始的，如图 17－32 所示。但是接收表面上看似乎是个被动的行为，其发起方应该是底层的驱动，应用进程反而是被调用的一方，其实不尽然，接收是个特殊的过程，分为两部分：表达接收意愿和向上投递数据报文。我们先来看看怎样向上投递。

1）向上投递数据报文

我们先来看投递数据报文的过程。

AFD 的 AfdPacketSocketReadData 函数是投递 UDP 数据包的具体执行者，其执行过程如下：

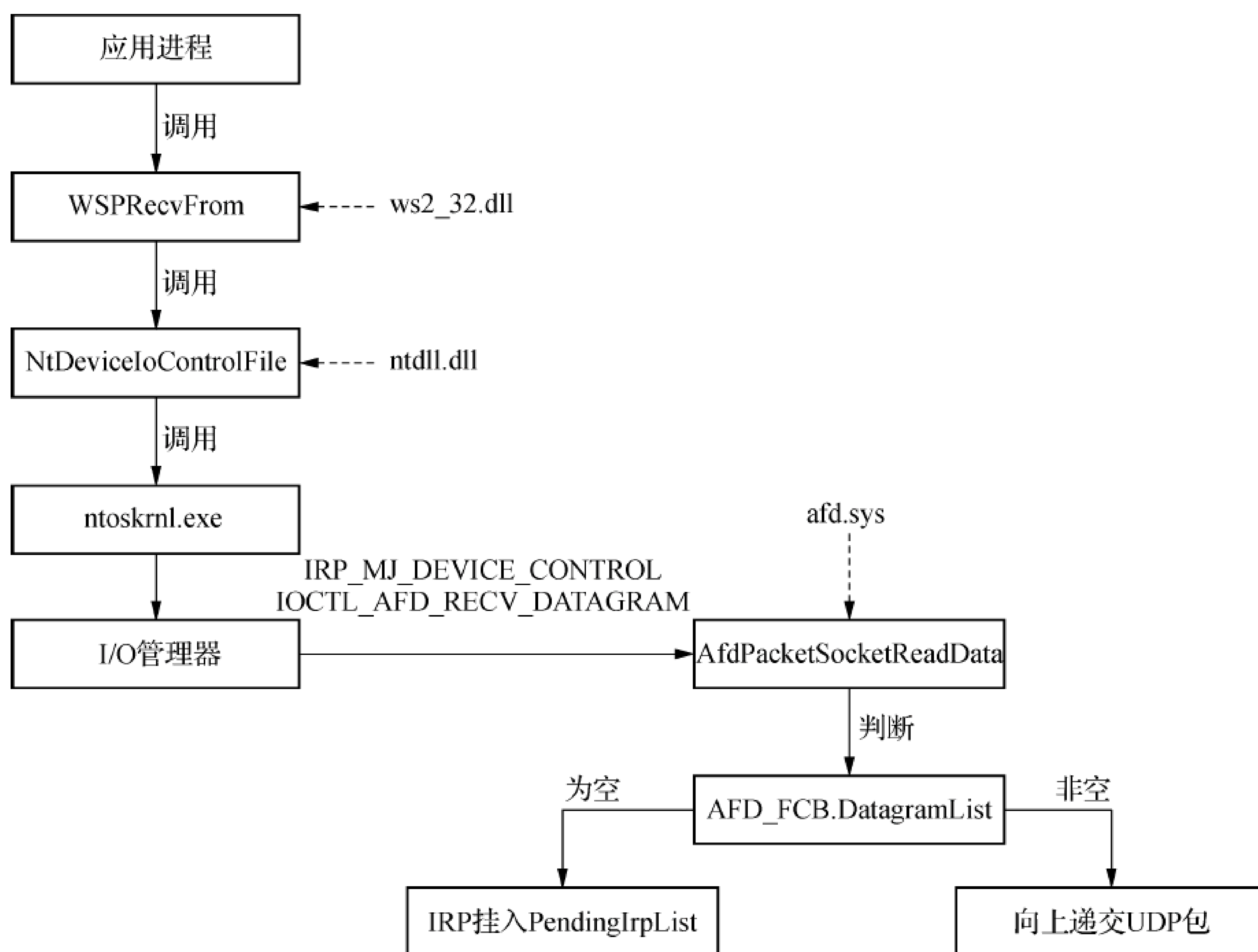


图 17－32　UDP 包的接收过程

（1）分配内存描述符表（MDL），将 AFD_RECV_INFO_UDP 数据结构映射到内核态空间，这个数据结构的具体域是这样的：

```
typedef struct_AFD_RECV_INFO_UDP
{
    PAFD_WSABUF BufferArray;          //存储接收的内容
    WINDOWS::ULONG BufferCount;
    WINDOWS::ULONG AfdFlags;
    WINDOWS::ULONG TdiFlags;
    WINDOWS::PVOID Address;           //PSOCKADDR_IN 结构，存储 IP 地址和端口号
    WINDOWS::PINT AddressLength;
}AFD_RECV_INFO_UDP, * PAFD_RECV_INFO_UDP;
```

AFD_RECV_INFO_UDP 用于 UDP 包的接收，其 BufferArray 域是接收缓冲区，这是个 AFD_WSABUF 结构的指针，在 IRP 中有域值与其对应绑定，可采用方法 LockRequest 将 IRP 与 AFD_RECV_INFO_UDP 结构（其实是 AFD_WSABUF）绑定。

（2）判断 AFD_FCB 的 DatagramList 队列中是否已经存在来自于下层的 UDP 包。

➢ 如果存在则立即摘除一个 UDP 包。UDP 包的结构是 AFD_STORE_DATAGRAM，将这个结构的内容复制到用户态空间缓冲区（通过 SatisfyPacketRecvRequest 方法实现），用户态空间缓冲区就是 IRP 中带下来的 AFD_WSABUF 结构，通过 MDL 将结构内容映射过去就好了。

➢ 如果不存在则将 IRP 挂入未完成队列中，这是通过 LeaveIrpUntilLater 方法实现的。这里的未完成队列是 AFD_FCB 的队列数组 PendingIrpList 的 FUNCTION_RECV 元

素,表示有相应的接收操作没有完成。

需要说明的是,投递 UDP 包也分为两个部分,以 AFD_FCB 的 DatagramList 为界,下层驱动将接收到的 UDP 包挂载到这个队列中,上层驱动也是在这个队列中提取 UDP 包并向上投递。因此 DatagramList 就像一个中转站一样传递着收到的报文。

显而易见,上述过程描述的是上层驱动对 UDP 包的提取,至于下层驱动对 UDP 包的挂载投递则是通过 tcpip. sys 中的 DGDeliverData 方法完成的。

在“协议驱动”一节中,我们描述了 tcpip. sys 的 LanReceiveWorker 的执行过程,这是在内核工作线程中执行的函数。其中执行到 ProtocolTable 表的 UDPReceive 时,通过调用 DGDeliverData 将 UDP 包上交到 AFD。

前文讲过,地址文件 ADDRESS_FILE 结构代表 socket,也就是说,每个 ADDRESS_FILE 都代表了协议、本地地址、本地端口这个三元组。对于 TCP 来说,它代表了客户端/服务端的地址和端口;对 UDP 的发送来说,它代表了打开的本地地址和一个打开的 UDP 端口(源地址和端口)。那么对于 UDP 的接收来说,ADDRESS_FILE 代表了 UDP 包的目的地址和端口。

DGDeliverData 首先获取目的端的 ADDRESS_FILE,判断 ADDRESS_FILE 的 ReceiveQueue 是否为空:

- 如果 ReceiveQueue 为空,则表明目前尚未有 UDP 端口开启或者尚未有端口有接收意愿,那么接下来判断 ADDRESS_FILE 的接收事件处理器(ReceiveDatagramHandler 函数)是否为空。接收事件处理器是由上层驱动注册的。
  - ReceiveDatagramHandler 为空,则表明这个 ADDRESS_FILE 确实没有一丁点的接收意愿,此时回复 ICMP 包给发送端。
  - ReceiveDatagramHandler 不空,则执行 ReceiveDatagramHandler。
- 如果 ReceiveQueue 不为空,则表明上层程序已经表达了接收意愿,有了意愿就要满足,但还不确定端口是否匹配。此时要比较一下 ReceiveQueue 中的每个有接收意愿的端口是否与 ADDRESS_FILE 的端口匹配,如果遇到了匹配的,就将接收到的 UDP 包中的 Buffer 拷贝到接收意愿(DATAGRAM_RECEIVE_REQUEST 结构)的 Buffer 中,并调用 DATAGRAM_RECEIVE_REQUEST 的完成函数 UDPReceiveComplete 来通知上层。

完成函数 UDPReceiveComplete 的执行过程如下:

① 创建一个 AFD_STORED_DATAGRAM 数据结构,将接收意愿中的 Buffer 拷贝到该数据结构的缓冲区中,并挂入 AFD_FCB 的 DatagramList 中。

② 判断 AFD_FCB 的 DatagramList 和 PendingIRPList[FUNCTION_RECV]是否为空,如果都不为空,则从 DatagramList 中获取一个 AFD_STORED_DATAGRAM 并从 PendingIRPList 中获取一个 IRP,调用 SatisfyPacketRecvRequest 回调给用户态空间缓冲区。

③ 通过 TdiReceiveDatagram 继续挂入一个新的接收意愿(DATAGRAM_RECEIVE_REQUEST)。

从这个完成函数我们可以看出：

➢ tcpip. sys 的 DGDeliverData 与接收意愿中的完成函数 UDPReceiveComplete 共同完成了 UDP 包投递的下层部分。

➢ 接收意愿是随着 UDP 包的上报而不断消耗的，因此要在完成函数执行过程中补充，即回调上去一个 UDP 包就补充一个接收意愿。

2）表达接收意愿

下面我们来看怎样表达接收意愿。TdiReceiveDatagram 方法是用来下传接收意愿的，其最终调用 UDPReceiveDatagram，如图 17－33 所示。接收意愿的首次下发过程如下：

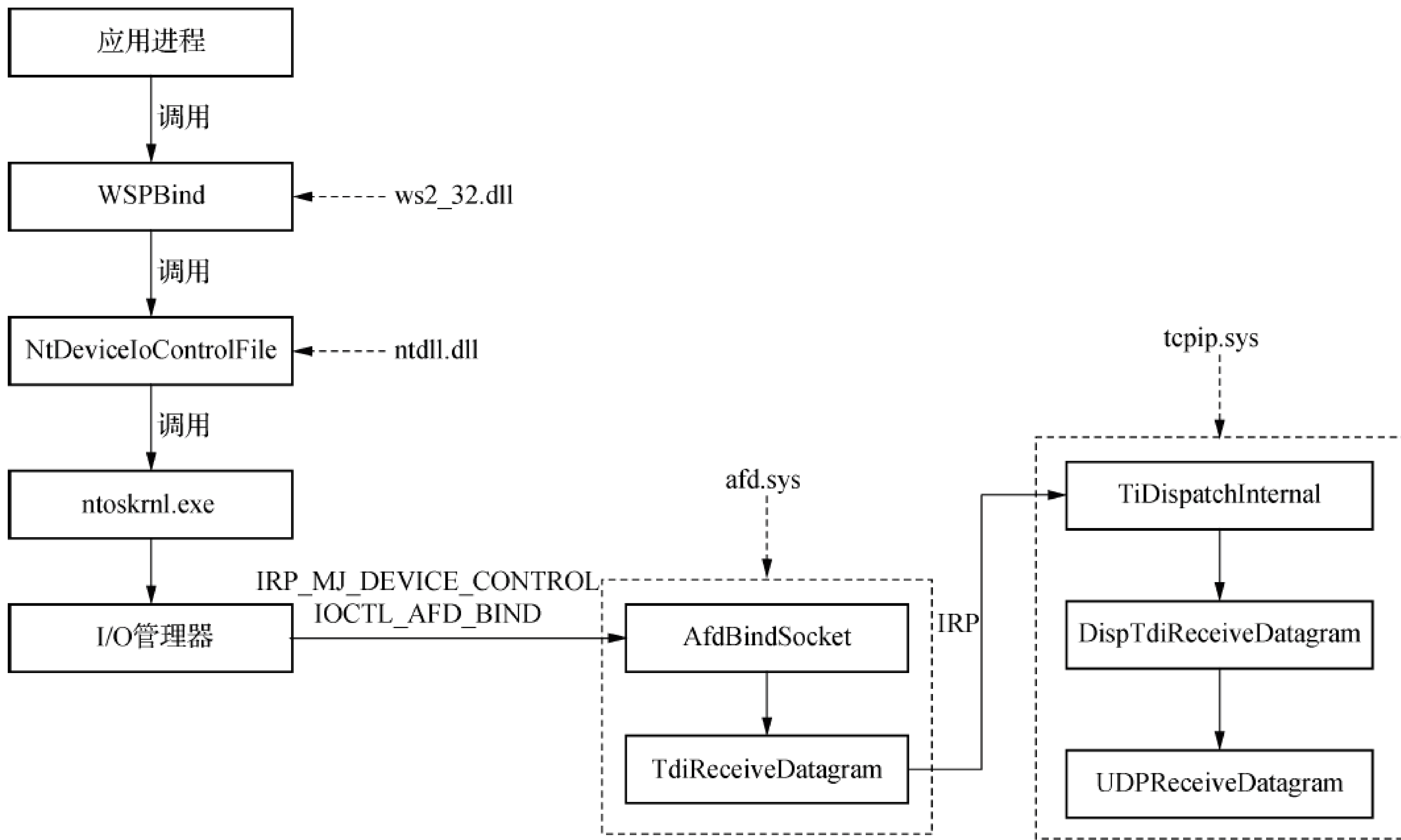


图 17－33　接收意愿的首次下发过程

（1）创建一个新的 DATAGRAM_RECEIVE_REQUEST，并初始化其中的成员变量。

（2）将 DATAGRAM_RECEIVE_REQUEST 挂入 ADDRESS_FILE 的 ReceiveQueue。

非常简单的意愿表达。通常是在绑定端口的时候下发第一个接收意愿，在接收到 UDP 包并调用完成函数的时候下发后续的接收意愿，以此来驱动源源不断的接收过程。

17.5　TDI/TDX 框架

1．TDI

TDI（Transport Driver Interface，传输驱动接口）是 Windows 系统中 AFD 与协议驱动之间的接口规范，其映像文件为 tdi. sys。但是在 Windows Vista 及之后的版本中，TDI 不再作为 AFD 与协议驱动的接口规范，而代之以 TLNPI（Transport Layer Network Provider Interface，传

输层网络提供者接口)、Windows 过滤平台和 WSK(WinSock Kernel)来作为新的接口规范，如图 17－34 所示，因此我们在本节不打算详细介绍 TDI。

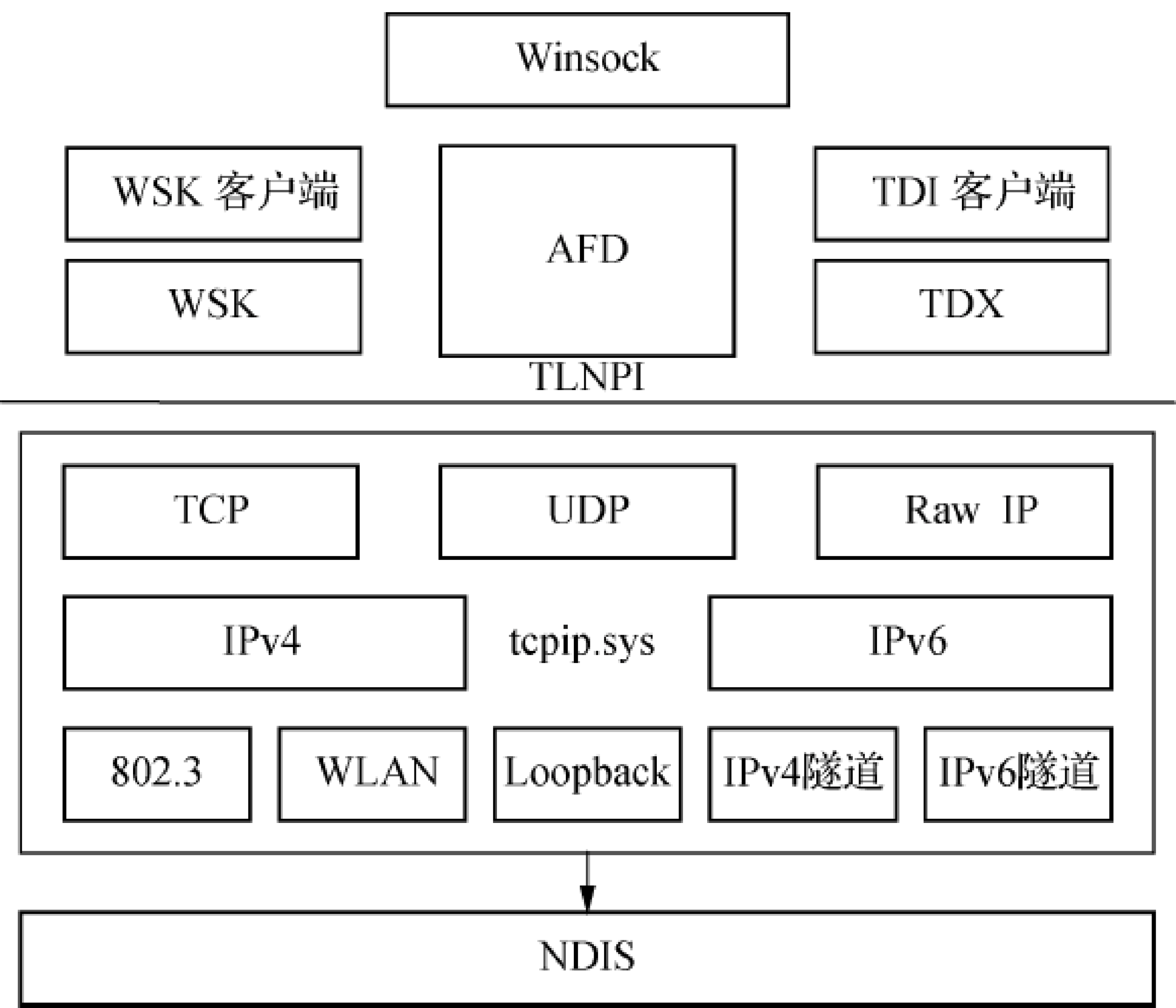


图 17－34　Windows Vista 及之后的网络协议栈体系结构

TDI 分为 TDI 传输客户端和 TDI 传输服务端。一般我们将 AFD 作为 TDI 传输客户端，意味着 AFD 要使用 TDI 提供的服务，将网络 API 转换成 IRP；将协议驱动作为 TDI 传输服务端，意味着协议驱动是 TDI 服务的提供者。

从图 17－35 可以看出，TDI 模块既依赖内核(ntoskrnl. exe)，也依赖 NDIS 模块。tdi. sys 的导出函数如图 17－36 所示。

tdi.sys

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntoskrnl.exe	42	0000A0E8	00000000	00000000	0000A0D4	00005000
NDIS.SYS	3	0000A240	00000000	00000000	0000A0C8	00005158

图 17－35　tdi. sys 依赖的模块

1）TDI 传输客户端

TDI 传输客户端采用地址对象来代表本地的地址端点(socket)，包括 TCP 和 UDP 两种通信方式中的地址；但是目的端的地址端点(socket)是采用连接对象来表示的，且仅限于 TCP 通信方式，因为 UDP 通信方式中没有连接的概念。图 17－37 中的 Address Object(地址对象)和 Connection Object(连接对象)都是用于表示 socket 的，且都是 FILE_OBJECT 类型，因此也都是由对象管理器创建的。

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
0000001C	00002BE0	001B	00009488	TdiDefaultConnectHandler
0000001D	00002C30	001C	000094A1	TdiDefaultDisconnectHandler
0000001E	00002C30	001D	000094BD	TdiDefaultErrorHandler
0000001F	00002BD0	001E	000094D4	TdiDefaultRcvDatagramHandler
00000020	00002BD0	001F	000094F1	TdiDefaultRcvExpeditedHandler
00000021	00002BD0	0020	0000950F	TdiDefaultReceiveHandler
00000022	00002C30	0021	00009528	TdiDefaultSendPossibleHandler
00000023	00004C30	0022	00009546	TdiDeregisterAddressChangeH...
00000024	00004CB0	0023	00009568	TdiDeregisterDeviceObject
00000025	00004A60	0024	00009582	TdiDeregisterNetAddress
00000026	00004C30	0025	0000959A	TdiDeregisterNotificationHandler
00000027	00004880	0026	000095BB	TdiDeregisterPnPHandlers
00000028	00004440	0027	000095D4	TdiDeregisterProvider
00000029	00004550	0028	000095EA	TdiEnumerateAddresses
0000002A	000025D0	0029	00009600	TdiGet9FTriageBlock
0000002B	000026F0	002A	00009614	TdiInitialize
0000002C	00001470	002B	00009622	TdiMapUserRequest
0000002D	00009758	002C	00009634	TdiMatchPdoWithChainedRecei...
0000002E	00002710	002D	00009659	TdiOpenNetbiosAddress
0000002F	00002330	002E	0000966F	TdiPnPPowerComplete

图 17－36　tdi. sys 的导出函数

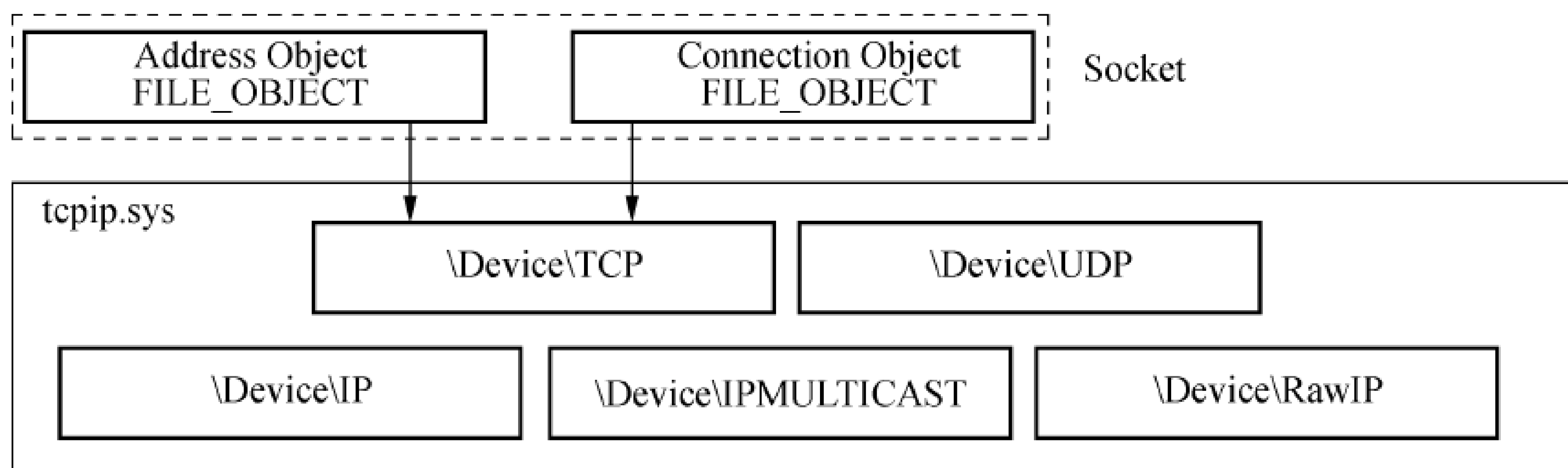


图 17－37　TDI 传输客户端的地址对象

2）TDI 传输服务端

TDI 传输客户端通过 IRP 与 TDI 传输服务端之间通信，这主要指命令的下发；TDI 传输服务端采用事件回调的方式向 TDI 传输客户端通知操作结果，如图 17－38 所示。例如 TCP 方式通信时的连接操作，作为发起者，TCP 客户端通过 IRP（TDI 主功能码为 IRP_MJ_DEVICE_CONTROL，TDI 副功能码为 TDI_CONNECT）通知己方一侧的 TDI 传输客户端，使 TDI 传输客户端向对端发起 TCP 连接，如图 17－39 所示。当对端（TCP 服务端）接受了连接

请求,会向 TCP 客户端的一侧返回接受通知(Accept),因此 TCP 客户端一侧的 TDI 传输服务端向本侧的 TDI 传输客户端返回 TDI_EVENT_CONNECT 的事件通知,如图 17-40 所示。

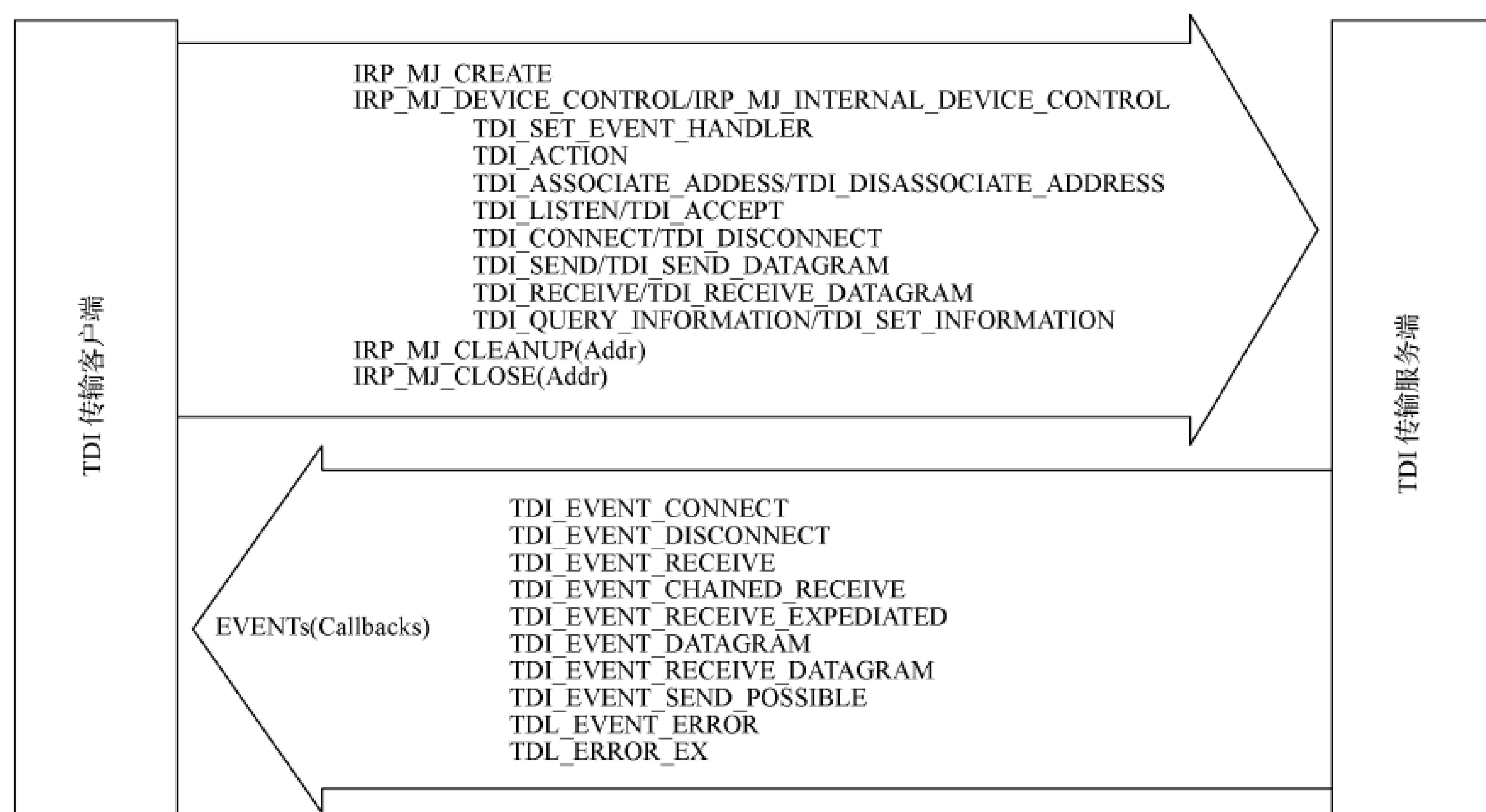


图 17-38　TDI 传输客户端与 TDI 传输服务端之间的交互(图片来自 CSDN)

TDI 副功能码与 TDI 传输客户端的 IRP 之间也是一一对应的,它与构造函数的关系如表 17-1 所示。TDI 不但有副功能码,还有若干回调事件,这些回调事件用于通知 AFD 驱动模块 TCP 握手过程中的某些状态,其定义如表 17-2 所示。

表 17-1　TDI 副功能码与 TDI 传输客户端的 IRP 构造函数之间的对应关系

请求	构造函数	在 AFD/TCPIP 中的应用
TDI_ASSOCIATE_ADDRESS	TdiBuildAssociateAddress()	将 TCP socket 的地址与连接对象关联起来
TDI_DISASSOCIATE_ADDRESS	TdiBuildDisassociateAddress()	将先前地址与连接对象的关联解除
TDI_CONNECT	TdiBuildConnect()	初始化三次握手(作为客户端)
TDI_LISTEN	TdiBuildListen()	服务端处于监听状态
TDI_ACCEPT	TdiBuildAccept()	接受客户端发来的连接请求
TDI_DISCONNECT	TdiBuildDisconnect()	与连接对端断开连接
TDI_SEND	TdiBuildSend()	向连接对端发送流数据(TCP 方式)
TDI_RECEIVE	TdiBuildReceive()	从连接对端接收数据流(TCP 方式)
TDI_SEND_DATAGRAM	TdiBuildSendDatagram()	向连接对端发送数据报(UDP)
TDI_RECEIVE_DATAGRAM	TdiBuildReceiveDatagram()	从连接对端接收数据报(UDP)

续表 17－1

请求	构造函数	在 AFD/TCPIP 中的应用
TDI_SET_EVENT_HANDLER	TdiBuildSetEventHandler( )	为地址对象注册/注销事件回调
TDI_QUERY_INFORMATION	TdiBuildQueryInformation( )	从 TCP/IP 协议栈查询 MIB 信息
TDI_SET_INFORMATION	TdiBuildSetInformation( )	该请求暂时不被 tcpip. sys 支持
TDI_ACTION	TdiBuildAction( )	该请求暂时不被 tcpip. sys 支持

表 17－2　TDI 事件回调的定义

事件	在 AFD/TCP 中的用途
TDI_EVENT_CONNECT	通知 afd. sys 上行的 TCP SYN 报文
TDI_EVENT_DISCONNECT	通知 afd. sys. 上行的 TCP FIN 报文
TDI_EVENT_ERROR	tcpip. sys 暂不支持
TDI_EVENT_RECEIVE	通知 afd. sys 上行的 TCP 数据段
TDI_EVENT_RECEIVE_DATAGRAM	通知 afd. sys 上行的 UDP 报文
TDI_EVENT_RECEIVE_EXPEDITED	通知 afd. sys 上行的 TCP URGENT 报文(带外数据)
TDI_EVENT_SEND_POSSIBLE	tcpip. sys 暂不支持
TDI_EVENT_CHAINED_RECEIVE	回调基于 NDIS_BUFFER 描述的 TCP/IP 数据副本
TDI_EVENT_CHAINED_RECEIVE_DATAGRAM	tcpip. sys 暂不支持
TDI_EVENT_CHAINED_RECEIVE_EXPEDITED	tcpip. sys 暂不支持
TDI_EVENT_ERROR_EX	向 afd. sys 报告目的的地址不可达

从图 17－40 可以看出,使用 TDI 之前需要设置事件通知回调函数,而创建 socket、绑定 socket、创建连接、断开连接等均在 TDI 的主副功能码中有对应的含义。

2. TDX

由于 Windows Vista 版本后微软不再支持 TDI,转而采用基于 TLNPI 的技术,而既有的大量基于 TDI 标准开发的 TDI 传输客户端驱动组件需要一个过渡组件进行适配,以兼容 TLNPI,因此 TDX 便应运而生。TDX 是 TDI 的适配组件,其上边沿仍然支持 TDI 接口,而下边沿支持 TLNPI 标准,完美解决了 TDI 向 TLNPI 转变过程中的兼容性问题。这再次证明了“Any problem in computer science can be solved by a middle layer(任何计算机领域的问题都可以通过引入一个中间层解决)”这句箴言。

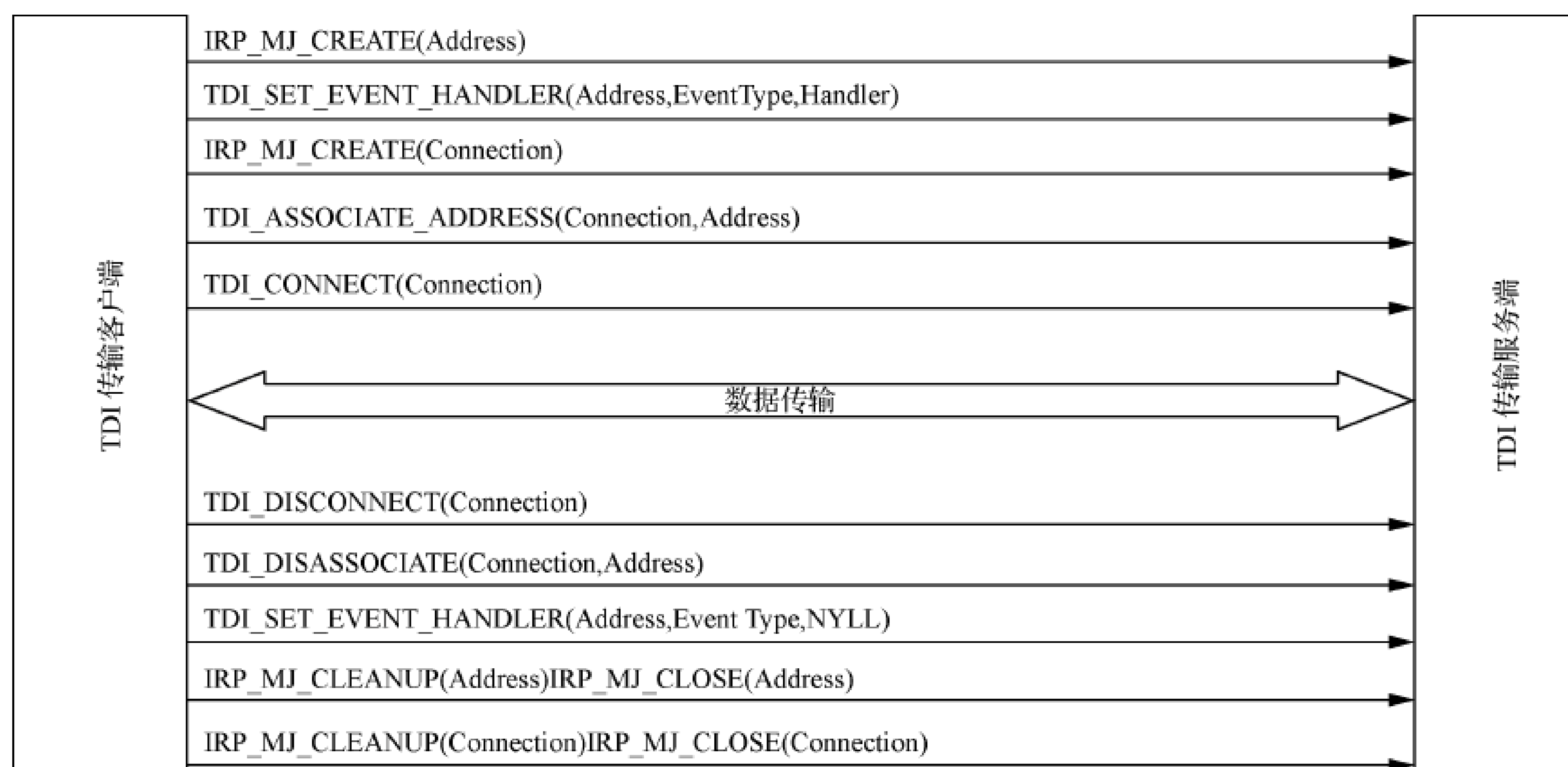


图 17－39　TCP 方式通信时 TDI 传输客户端与 TDI 传输服务端的交互（TCP 客户端一侧）（图片来自 CSDN）

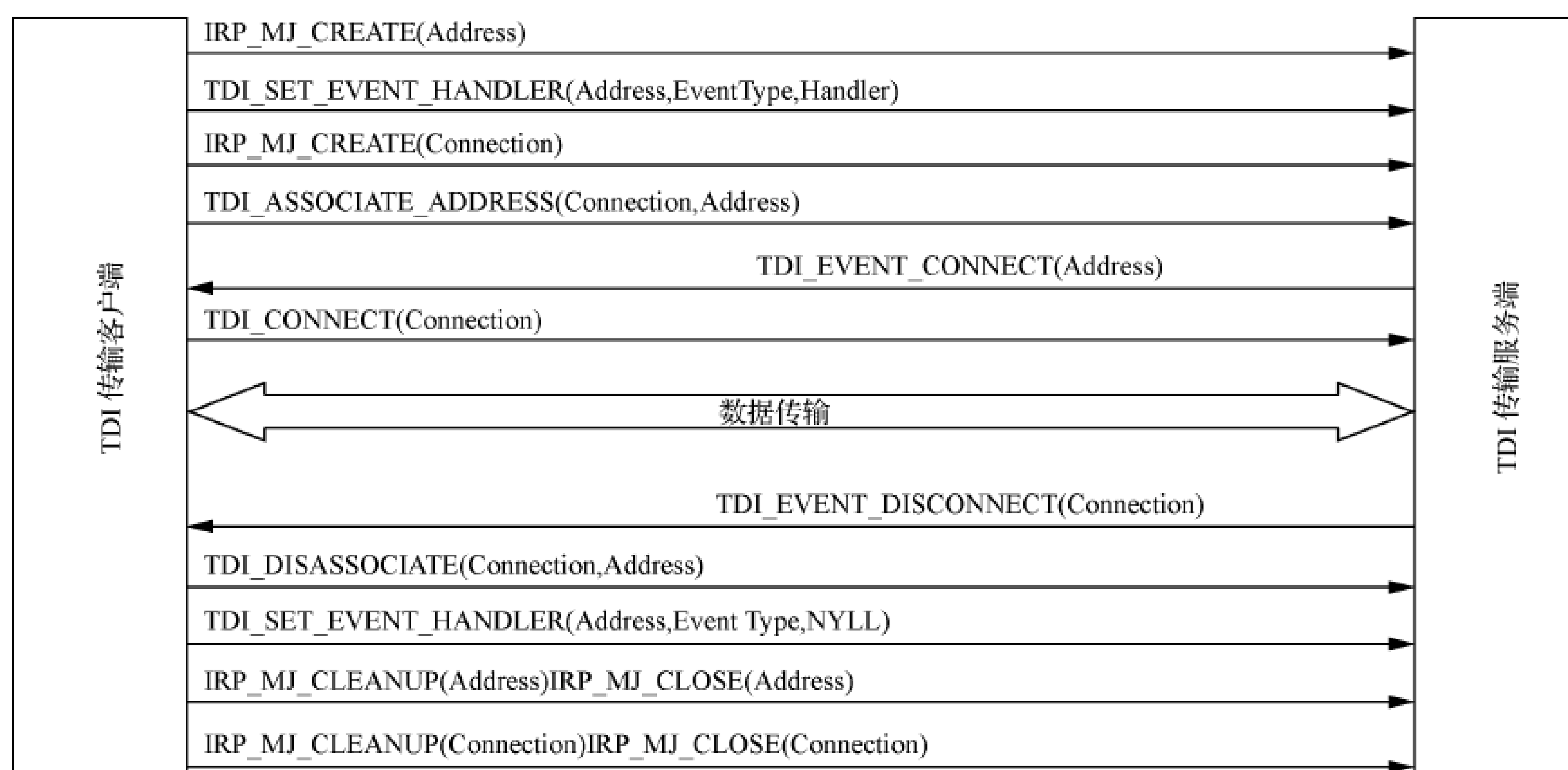


图 17－40　TCP 方式通信时 TDI 传输客户端与 TDI 传输服务端的交互（TCP 服务端一侧）（图片来自 CSDN）

TDX 创建了若干个代表传输协议（四层协议）的设备对象，如图 17－41 左半部分所示，使 TDI 传输客户端可以获得一个代表某种协议的文件对象，并且利用 IRP 向协议驱动发送 I/O 请求。

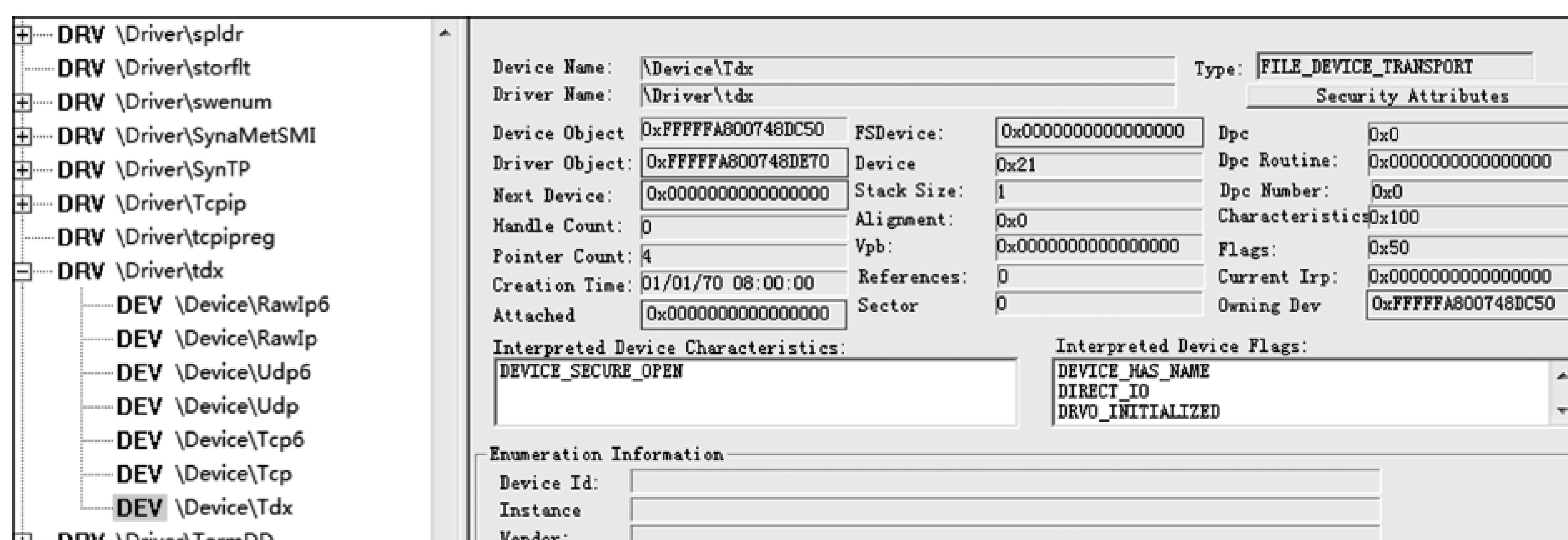


图 17－41　TDX 创建的设备对象

TDX 的映像文件是 tdx. sys，其依赖库如图 17 - 42 所示，其架构如图 17 - 43 所示。

tdx.sys

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntoskrnl.exe	74	0001F120	00000000	00000000	0001F110	00019000
NETIO.SYS	19	0001F378	00000000	00000000	0001F104	00019258
TDI.SYS	9	0001F418	00000000	00000000	0001F0FC	000192F8
NDIS.SYS	1	0001F468	00000000	00000000	0001F0F0	00019348

图 17 - 42　tdx. sys 的依赖库

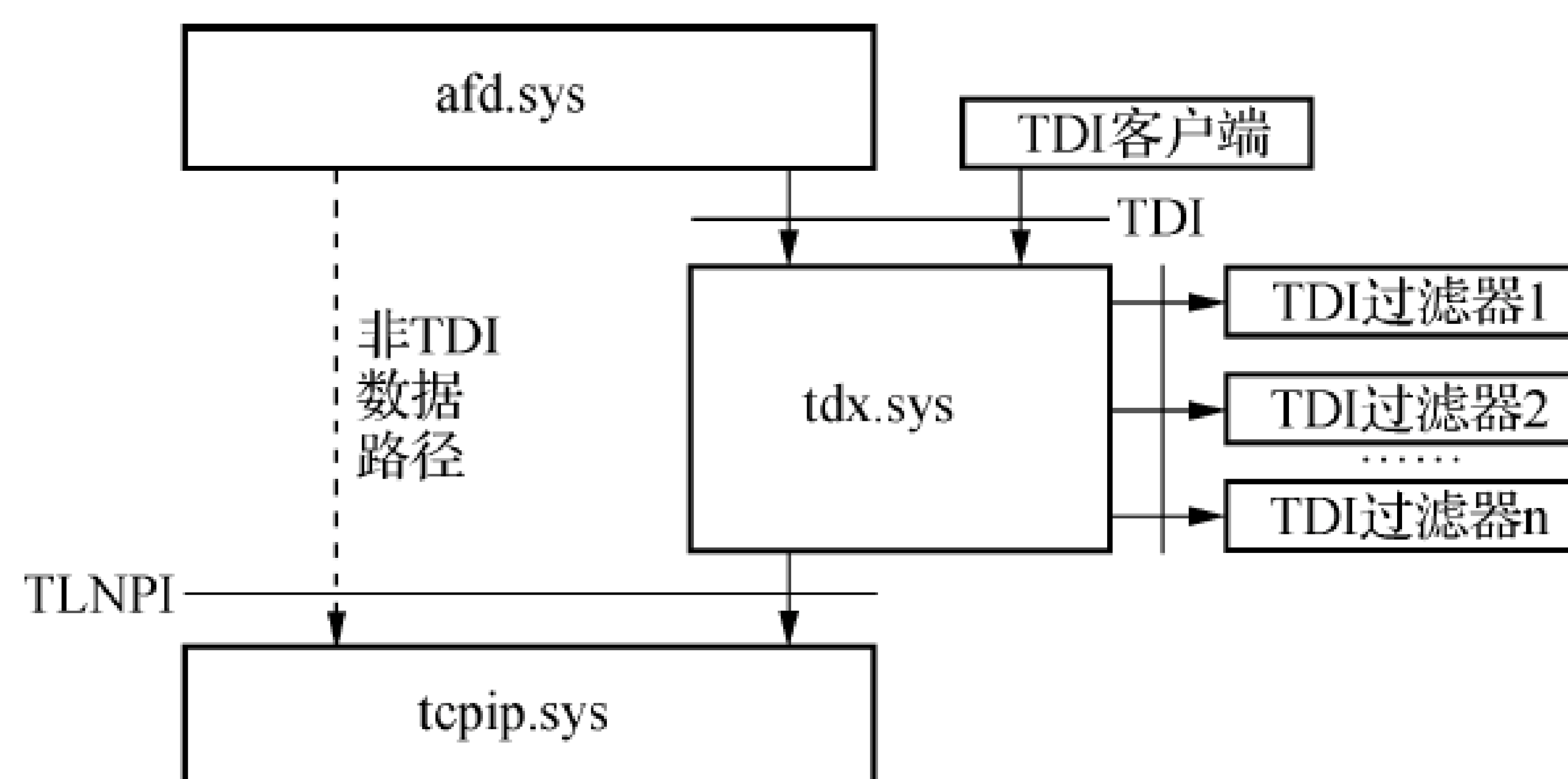


图 17 - 43　TDX 架构视图

17.6　WSK 框架

WSK（WinSock Kernel，WinSock 内核）是为了使内核态驱动程序能够方便使用 socket API 而实现的内核库，是内核态的网络编程接口。WSK 的出现取代了传统的 TDI，其具有更好的性能、易用性、安全性和伸缩性。有了 WSK 后内核进程可以不必再使用 TDI 进行网络通信，而是像用户态进程使用 socket API（ws2_32. dll）那样方便地进行网络编程了。

WSK 的总体架构如图 17 - 44 所示。其中 WSK 使用 NMR（Network Module Register，网络模块注册器）堆叠到传输设备（例如 Device\TCP 等）上去，也支持从堆叠中卸载。



图 17 - 44　WSK 架构视图

WSK 被分为 WSK 子系统和 WSK 客户端两个角色。与 TDI 类似,WSK 客户端就是 WSK 服务的使用者,它可以通过 NMR 发布的注册函数向 WSK 子系统注册或注销,当然也可以通过 WSK 子系统(实际上是在 AFD 中实现的)自身发布的注册 API(WskRegister、WskDeregister 等)注册或注销,如图 17-45 所示。同样,WSK 子系统反调 WSK 客户端的事件通知回调函数来通知完成事件(例如 WskAcceptEvent、WskReceiveEvent、WskDisconnectEvent 等)。

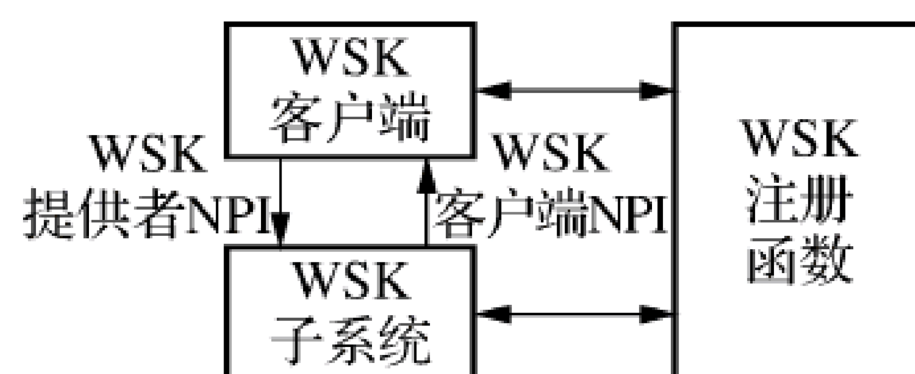


图 17-45　WSK 的注册

17.7　WFP 框架

WFP(Windows Filtering Platform,Windows 过滤平台)是 Windows Vista 及后续版本中新增的一套系统服务框架,用于网络数据包的过滤,以代替较早版本的 TDI 框架、NDIS 框架等。WFP 本身不实现网络包过滤功能,它只是一套框架,诸如防火墙、入侵检测系统、网络流量监测系统等基于网络包过滤的业务系统均可以通过 WFP 框架来实现。

WFP 框架分为用户态过滤引擎(UM Filtering Engine)和内核态过滤引擎(KM Filtering Engine)两大部分。其中用户态过滤引擎又叫作基础过滤引擎(Base Filtering Engine,BFE),BFE 与内核态过滤引擎交互,而第三方的应用又通过 C 风格的 API 或 RPC 接口与 BFE 交互。BFE 的接口封装在用户态模块 FWPUCLNT. DLL 中。WFP 框架整体架构如图 17-46 所示,FWPUCLNT. DLL 的导出函数则见图 17-47。

内核态过滤引擎是 WFP 框架的核心,无论第三方应用与 BFE 有着怎样的交互,BFE 都会将过滤请求透传至内核态过滤引擎中,真正实施过滤操作的是内核态过滤引擎,引擎通过“垫片”(shim)策略监测协议栈驱动中不同层次的网络数据。

如图 17-46 中左下方所示,在 tcpip. sys 中一共插入了 4 块垫片,从上到下分别为数据流层垫片(应用层的原始数据包)、应用层垫片、传输层垫片和网络层垫片,一层比一层更接近 OSI 模型的底层。这非常好理解:WFP 利用垫片策略机制在协议栈驱动的不同层次中分别插入“楔子”,以过滤不同层次的网络包。

通过垫片获取到网络包数据后,通过内核态过滤引擎提供的分类 API 将这些数据传送到 WFP 对应的分层中,例如 ALE 分层(应用层连接管理,处理 Bind、Connect、Accept、Listen 等 TCP 操作),而这些分层又可以通过 WFP 的调出接口 Callout 将这些数据导出到第三方系统中,如图 17-48 所示。WFP 框架为每一分层都设置了若干过滤挂钩点,框架使用者可以对这些过滤挂钩点设置不同类型的钩子函数以监控数据流。例如在 ALE 分层就针对 TCP 客户端设置了如下过滤挂钩点:

- **FWPM_LAYER_ALE_BIND_REDIRECT_V4**:对应 Windows 7 以上版本的 Bind 操作;
- **FWPM_LAYER_ALE_CONNECT_REDIRECT_V4**:对应 Windows 7 以上版本的

图 17－46　WFP 框架整体架构

Connect 操作：

- **FWPM_LAYER_OUTBOUND_TRANSPORT_V4**：SYN 下行到传输层；
- **FWPM_LAYER_OUTBOUND_IPPACKET_V4**：SYN 下行到网络层；
- **FWPM_LAYER_INBOUND_IPPACKET_V4**：SYN-ACK 上行到网络层；
- **FWPM_LAYER_INBOUND_TRANSPORT_V4**：SYN-ACK 上行到传输层；
- **FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4**：SYN-ACK 到达 ALE 分层，内核建立起连接，同时回复 ACK 包给服务端；

File: FWPUCLNT.DLL
- Dos 头部
- Nt 头部
  - 文件头部
  - 可选头部
    - 数据目录 [x]
- 段头部 [x]
- 输出目录
- 导入目录
- 资源目录
- 异常目录
- 重定位目录
- 调试目录
- Address Converter
- Dependency Walker
- 十六进制编辑
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00007D20	0000	0002AB77	FwpmCalloutAdd0
00000002	0000E854	0001	0002AB87	FwpmCalloutCreateEnumHandle0
00000003	0000E60C	0002	0002ABA4	FwpmCalloutDeleteById0
00000004	0000E734	0003	0002ABBB	FwpmCalloutDeleteByKey0
00000005	0000E968	0004	0002ABD3	FwpmCalloutDestroyEnumHand...
00000006	0000E8CC	0005	0002ABF1	FwpmCalloutEnum0
00000007	0000E6A8	0006	0002AC02	FwpmCalloutGetById0
00000008	0000E7CC	0007	0002AC16	FwpmCalloutGetByKey0
00000009	0000E9E8	0008	0002AC2B	FwpmCalloutGetSecurityInfoByK...
0000000A	0000EAFC	0009	0002AC4C	FwpmCalloutSetSecurityInfoByK...

图 17－47　FWPUCLNT. DLL 导出的函数

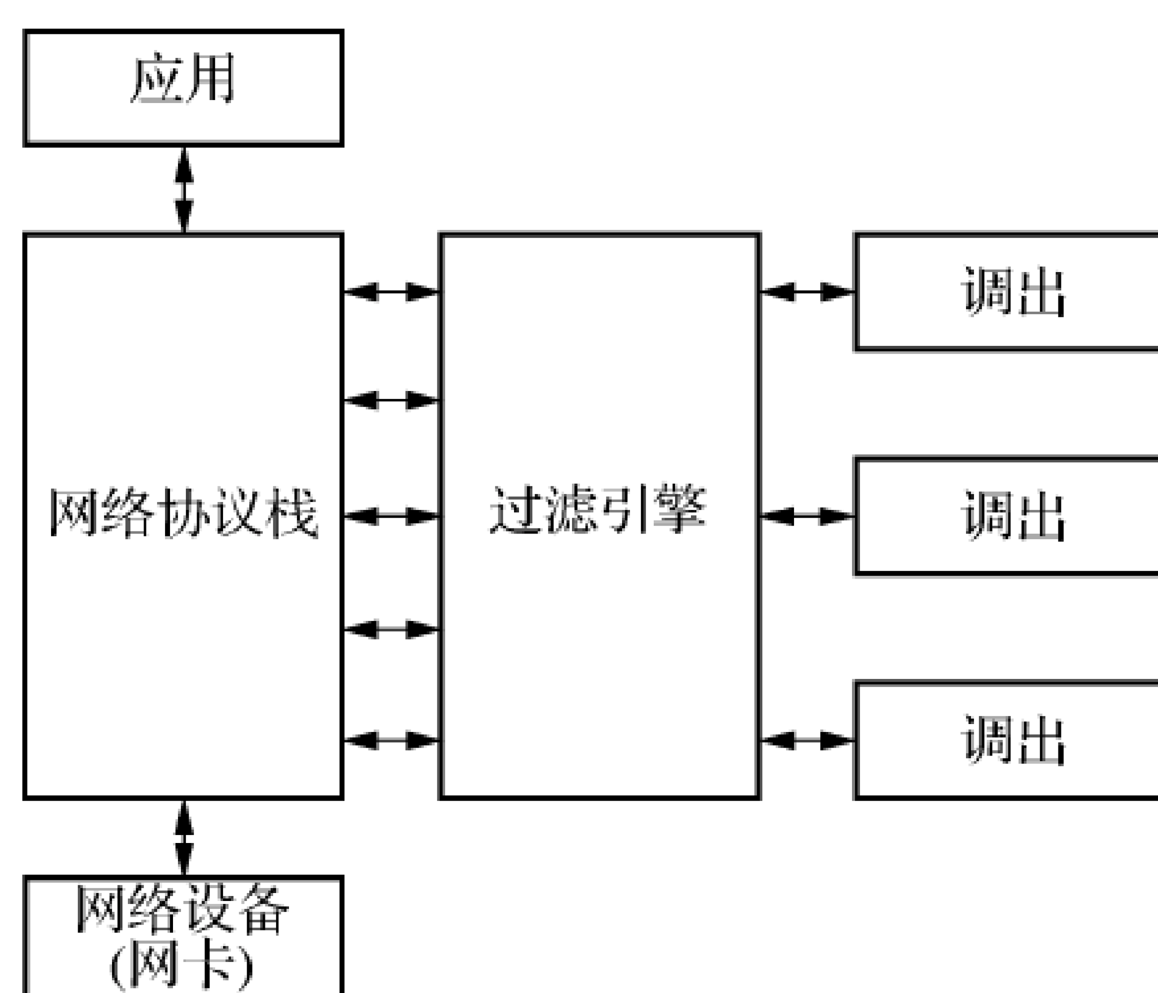


图 17－48　WFP 机制示意图

- **FWPM_LAYER_OUTBOUND_TRANSPORT_V4**：回复的 ACK 包下行到传输层；
- **FWPM_LAYER_OUTBOUND_IPPACKET_V4**：回复的 ACK 包下行到网络层。

图 17－49 显示的是 Windows 7 系统下 360 等软件利用 WFP 框架注册的过滤挂钩点。

1）内核态过滤引擎

内核态过滤引擎是整个 WFP 框架的核心，该引擎也分成若干层，分别对应了网络协议栈的不同层次，每一个分层又可以包含若干子层和子过滤器。内核态过滤引擎检测网络数据包是否命中子过滤器的规则，如果命中则会执行子过滤器中指定的动作（放行还是拦截）。对于命中了多个子过滤器规则的情况，WFP 还引入了过滤仲裁器，由仲裁器计算出最终的动作来交给内核态过滤引擎，内核态过滤引擎再把结果反馈给对应层次的垫片。

2）垫片

垫片的作用有两个：

- 截取协议栈驱动中各层的数据包并返回给过滤引擎；
- 将过滤引擎中对于网络数据包的处理结果返回给协议栈驱动，使之进行对应的放行/拦截操作。

SPI(网络连接劫持) | 网络连接 | WFP网络过滤 | IE相关 | 隐藏账户/克隆账户

CalloutID	FilterID	过滤层	名称	描述
256	65778	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	360netmon	360netmon
266	68149	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	AntiAttack	Callout that forward AntiAttack
258	65785	FWPM_LAYER_OUTBOUND_TRANSPORT_V4	360netmon	360netmon
259	65786	FWPM_LAYER_STREAM_V4	360netmon	360netmon
263	68152	47c9137a-7ec4-46b3-b6e448e926b1eda4	360AntiHijack	Callout that forward 360AntiHijack
260	65787	FWPM_LAYER_DATAGRAM_DATA_V4	360netmon	360netmon
264	68155	7021d2b3-dfa4-406e-afeb6afaf7e7efd	360AntiHijack	Callout that forward 360AntiHijack
284	68128	FWPM_LAYER_STREAM_V4	360AntiHijack	Callout that forward 360AntiHijack
262	68132	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	360AntiHijack	Callout that forward 360AntiHijack
260	65782	FWPM_LAYER_DATAGRAM_DATA_V4	360netmon	360netmon
267	68156	FWPM_LAYER_OUTBOUND_TRANSPORT_V4	360 AntiHacker Callout Outbound ...	360 AntiHacker Callout Outbound ...
258	65780	FWPM_LAYER_OUTBOUND_TRANSPORT_V4	360netmon	360netmon
261	65788	c6e63c8c-b784-4562-aa7da67cfcaf9a3	360netmon	360netmon
256	65783	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	360netmon	360netmon
268	68157	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	360 AntiHacker Flow Establish	360 AntiHacker Flow Establish
262	68133	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	360AntiHijack	Callout that forward 360AntiHijack
264	68154	7021d2b3-dfa4-406e-afeb6afaf7e7efd	360AntiHijack	Callout that forward 360AntiHijack
266	68148	FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	AntiAttack	Callout that forward AntiAttack
259	65781	FWPM_LAYER_STREAM_V4	360netmon	360netmon
265	68147	FWPM_LAYER_STREAM_V4	AntiAttack	Callout that forward AntiAttack
263	68153	47c9137a-7ec4-46b3-b6e448e926b1eda4	360AntiHijack	Callout that forward 360AntiHijack
257	65779	FWPM_LAYER_INBOUND_TRANSPORT_V4	360netmon	360netmon
284	68129	FWPM_LAYER_STREAM_V4	360AntiHijack	Callout that forward 360AntiHijack
257	65784	FWPM_LAYER_INBOUND_TRANSPORT_V4	360netmon	360netmon
265	68146	FWPM_LAYER_STREAM_V4	AntiAttack	Callout that forward AntiAttack

图 17－49　Windows 7 系统下设置的过滤挂钩点示例

垫片相当于网络协议栈中的“探针”，是 WFP 框架的组成部分，但对于 WFP 的编程者来说则是透明的。

3）调出接口

调出接口 Callout 也是 WFP 框架的组成部分，由若干回调函数构成。当网络数据包命中了某过滤器的规则时，如果该过滤器指定了调出接口，则调出接口中的回调函数就会被调用。调出接口包含三个回调函数：notifyFn、classifyFn 和 flowDeleteFn。

- **classifyFn**：若过滤器关联了调出接口，当规则命中时，内核态过滤引擎会回调该函数，WFP 框架的使用者可以借用该函数进一步处理网络数据包的信息（例如执行深度包检测、修改包头地址、做 NAT 等），也可以设置对数据包的放行/拦截操作。
- **notifyFn**：当过滤器被添加到内核态过滤引擎或从中移除时，WFP 框架调用该过滤器对应的调出接口的 notifyFn 函数，以便通知 WFP 框架的使用者过滤器的变化情况。
- **flowDeleteFn**：当一个网络数据流将要被终止时，WFP 框架调用 flowDeleteFn 函数来清理上下文。

4）过滤器

过滤器存在于内核态过滤引擎的各个分层中。WFP 框架内置了一部分过滤器供第三方模块使用，同时开发者也可以添加自己的过滤器。过滤器是规则与动作的组合，满足了规则就要执行相应的动作。在一个分层中可以存在多个过滤器，每个过滤器被赋予不同的权重。过滤器也可以关联调出接口，当规则被命中时，调出接口对应的回调函数被调用，可以在回调函数内对网络数据包做进一步的处理。

最后介绍一下 WFP 框架的工作流程，如下所示：





（1）一个网络包进入到协议栈驱动中（tcpip.sys）。

（2）协议栈驱动找到一个垫片，这是由若干个分层注册的垫片。

（3）该垫片调用该特定分层注册的 Classfication（分类） API 进行分类处理。

（4）在分类处理期间，过滤器开始工作，进行条件匹配，如果条件匹配，则调用对应的调出接口 classifyFn。

（5）垫片执行最终的过滤决定（放行还是拦截）。

对比 Linux 平台固有的 Netfilter 框架，Netfilter 框架只能工作在网络层，过滤拦截的都是 IP 包；而 WFP 框架可以工作在应用层、传输层、网络层甚至链路层（Windows 8.1 之后的版本），可以说是全栈方式的检测与拦截，功能比 Netfilter 框架更为强大。

17.8 WinPcap 框架

WinPcap 框架是 Windows 平台访问数据链路层的开源库，其特点是可以绕开 Windows 自带的协议栈驱动程序而进行网络数据包的捕获/发送，因此该框架常用于网络数据包监听和旁路化截取。我们日常使用的 Wireshark 抓包软件就是基于 WinPcap 框架实现的。

这里要注意，WinPcap 是通过旁路技术实现监听和截取的，因此不会改变既有数据包的内容，更不会阻塞既有数据包的传递，它只是将接收或发送的网络数据包做一份拷贝供自己分析使用，因此也可能会对系统的网络 I/O 性能造成一定的影响。

WinPcap 框架具有以下功能：

- 捕获原始数据包，无论它是发往某台机器的，还是在其他设备（共享媒介）上进行交换的。
- 在网络数据包到达某应用进程之前，根据用户指定的规则过滤数据包。
- 支持以可编程的方式将原始数据包通过网络发送出去。
- 支持以可编程的方式收集并统计网络流量信息。

WinPcap 框架由以下组件构成，三者的层次关系如图 17－50 所示。

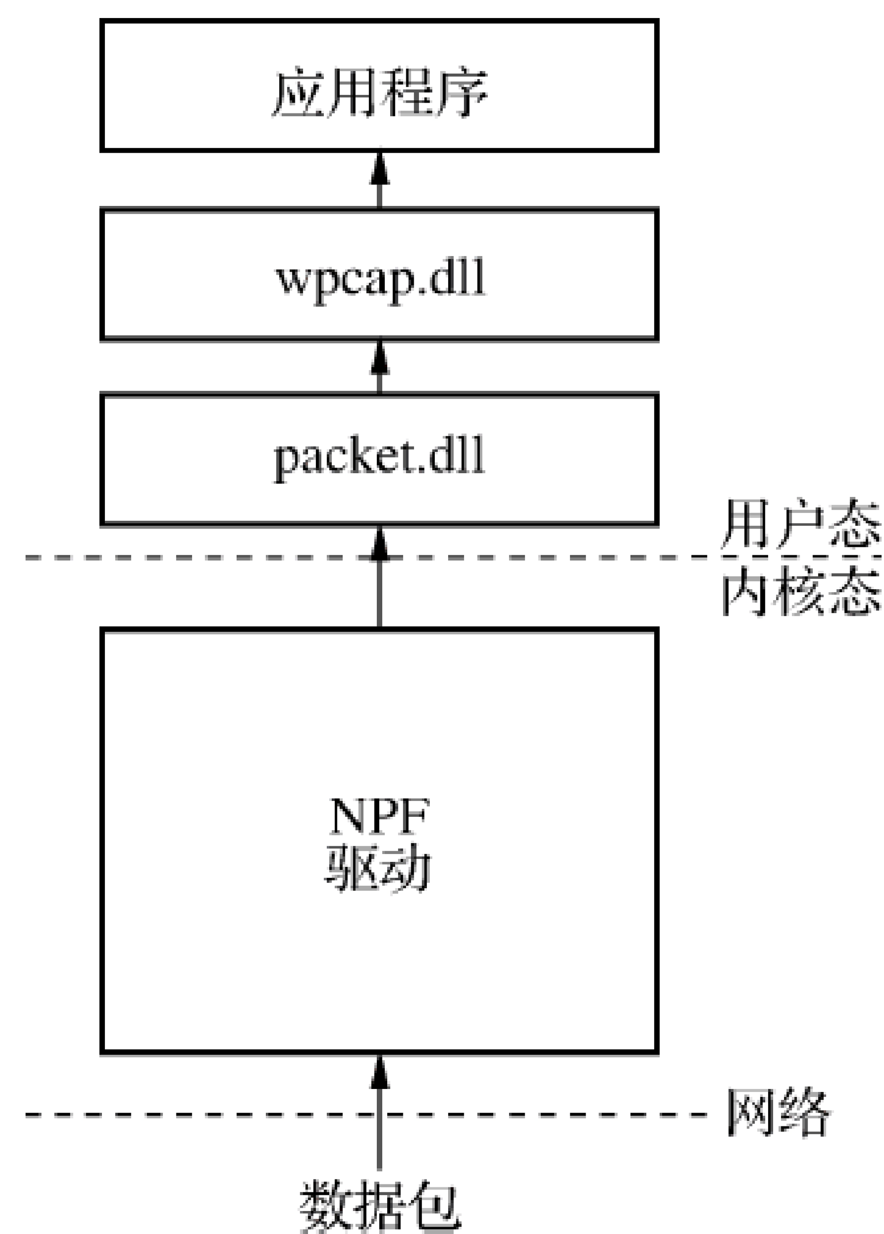


图 17－50　WinPcap 框架的组成

- **NPF（Netgroup Packet Filter，网络组包过滤器）驱动**：这是个内核态驱动，提供了数据包捕获与发送功能。
- **packet.dll**：这是个用户态支撑库，提供了用户态 API，用于直接访问 NPF 驱动，并向 wpcap.dll 提供功能支持。packet.dll 的导出函数如图 17－51 所示。

File: Packet.dll
Dos 头部
Nt 头部
文件头部
可选头部
数据目录 [x]
段头部 [x]
输出目录
导入目录
资源目录
异常目录
重定位目录
调试目录
Address Converter
Dependency Walker
十六进制编辑
Identifier
Import Adder
Quick Disassembler
Rebuilder
Resource Editor

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
0000000A	000063E0	0009	00026470	PacketGetNetInfoEx
0000000B	00006600	000A	00026483	PacketGetNetType
0000000C	00005CF0	000B	00026494	PacketGetReadEvent
0000000D	00005E60	000C	000264A7	PacketGetStats
0000000E	00005F00	000D	000264B6	PacketGetStatsEx
0000000F	00005370	000E	000264C7	PacketGetVersion
00000010	000057A0	000F	000264D8	PacketInitPacket
00000011	00005C70	0010	000264E9	PacketIsDumpEnded
00000012	00006760	0011	000264FB	PacketIsLoopbackAdapter
00000013	00006850	0012	00026513	PacketIsMonitorModeSupported
00000014	00005490	0013	00026530	PacketOpenAdapter

图 17－51　packet. dll 的导出函数

➢ **wpcap. dll**：这也是个用户态支撑库，为上层应用提供了最上层的 API 支持，其功能更强大。wpcap. dll 的导出函数如图 17－52 所示。

WinPcap 框架整体架构如图 17－53 所示。

File: wpcap.dll
Dos 头部
Nt 头部
文件头部
可选头部
数据目录 [x]
段头部 [x]
输出目录
导入目录
资源目录
异常目录
重定位目录
调试目录
Address Converter
Dependency Walker
十六进制编辑
Identifier
Import Adder
Quick Disassembler
Rebuilder
Resource Editor

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	0005FEF8	00060290	000600C8	000603A5
(nFunctions)	Dword	Word	Dword	szAnsi
00000059	00019740	0058	00061595	pcap_sendqueue_transmit
0000005A	00019780	0059	000615AD	pcap_set_buffer_size
0000005B	000197C0	005A	000615C2	pcap_set_datalink
0000005C	00019930	005B	000615D4	pcap_set_immediate_mode
0000005D	00019970	005C	000615EC	pcap_set_promisc
0000005E	000199B0	005D	000615FD	pcap_set_rfmon
0000005F	000199F0	005E	0006160C	pcap_set_snaplen
00000060	00019A30	005F	0006161D	pcap_set_timeout
00000061	00019A70	0060	0006162E	pcap_set_tstamp_precision
00000062	00019B00	0061	00061648	pcap_set_tstamp_type

图 17－52　wpcap. dll 的导出函数

WinPcap 框架中的 NPF 驱动通过向 NDIS 框架注册绑定了所有的网卡。注册成功后，NPF 的地位就与 TCP/IP 协议栈驱动的地位一样了，任何一个网卡接收到网络数据包后都可以向协议驱动和 NPF 驱动回调，NPF 就是这样借助 NDIS 框架和旁路机制实现了网络包的捕获，如图 17－54 所示。

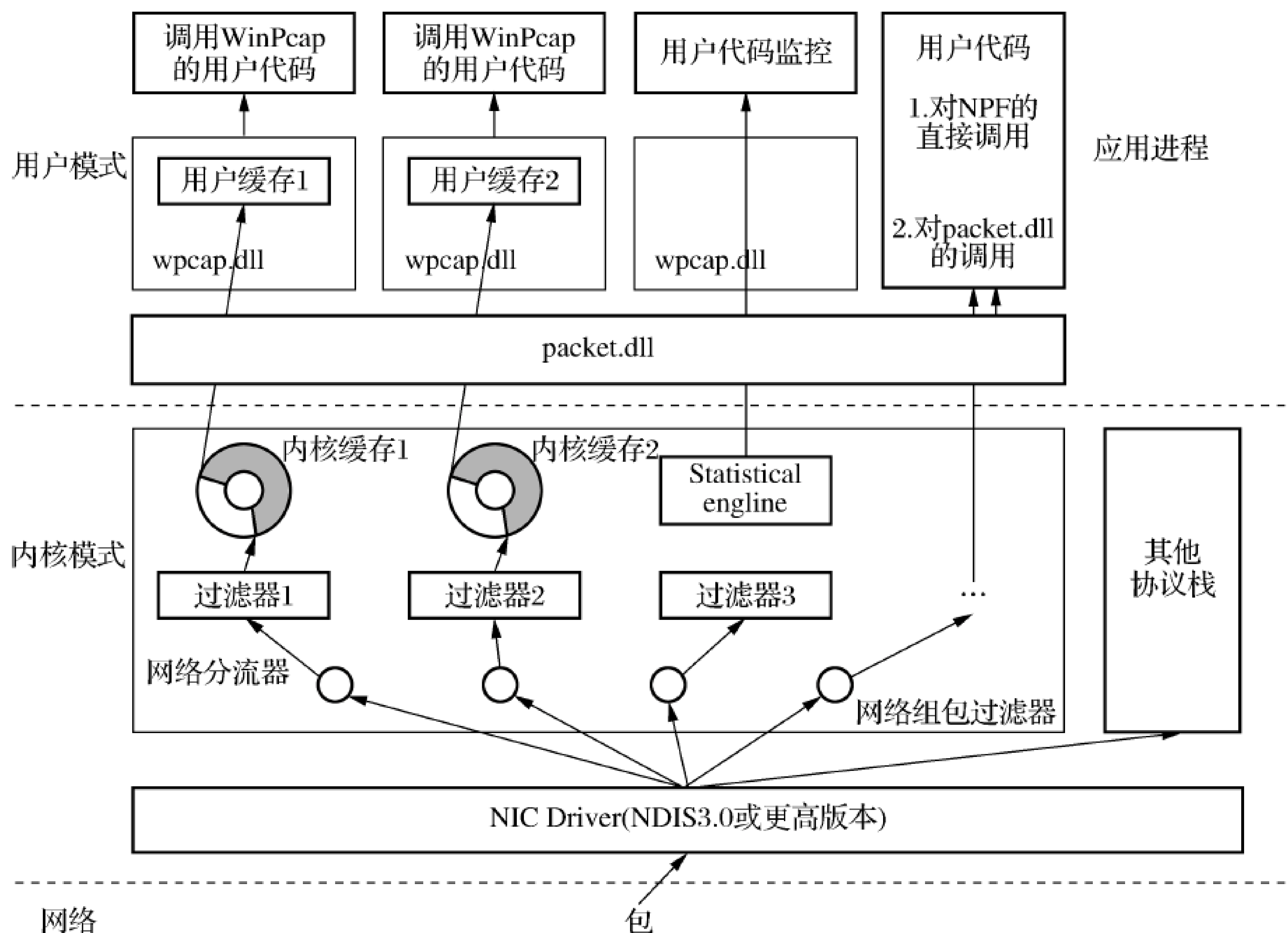


图 17－53　WinPcap 框架整体架构

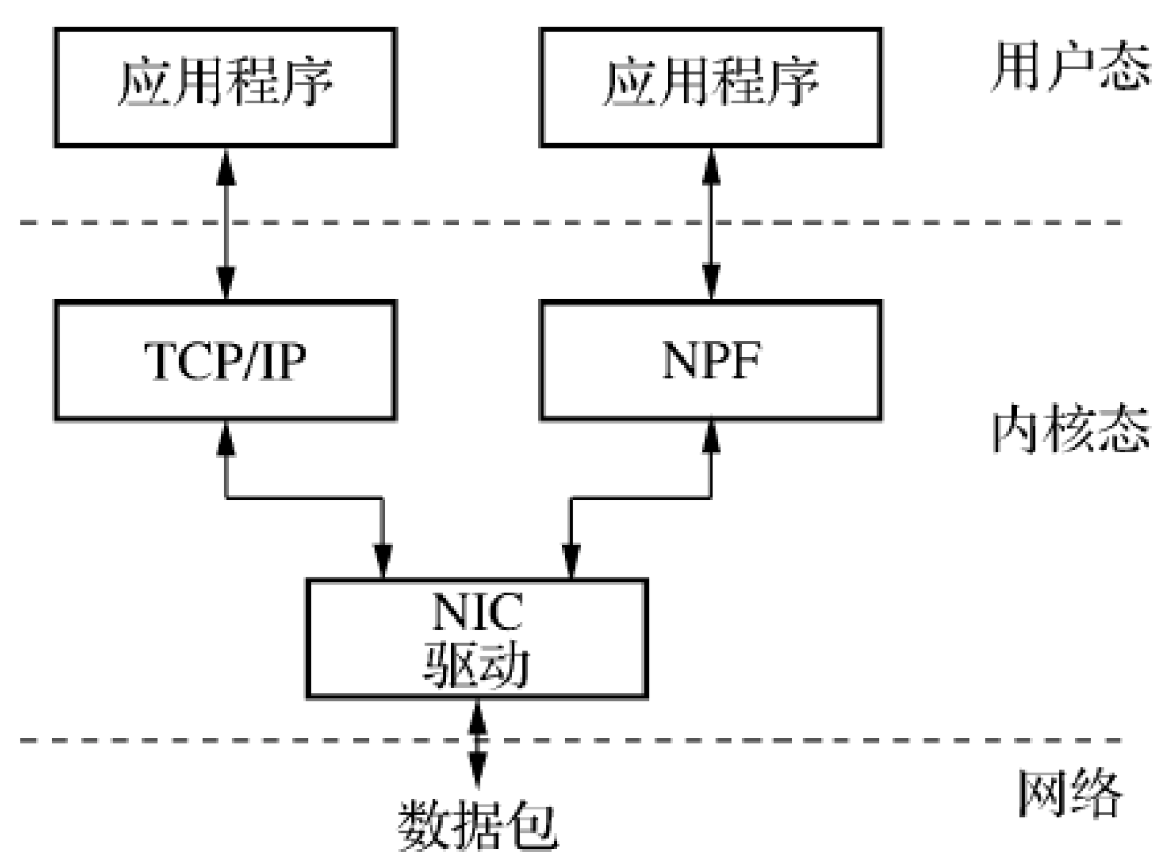


图 17－54　网络协议栈旁路机制

本章小结

本章介绍了 Windows 系统中的网络协议栈的各个驱动和框架。其中比较关键的是 NDIS，这是一个串联 NDIS 小端口驱动和协议驱动的接口框架，但不应该把这个框架看作扁平的一层，因为 NDIS 不仅对接了网卡驱动和协议驱动，也为整个网络协议栈提供了支撑接口。

TDI/TDX 则作为用户态支撑库与辅助功能驱动的接口，是 socket 机制的分水岭。TDI 之上就是我们熟知的 socket 库，之下则实现了 socket 机制比较底层的逻辑。

在此基础之上，还介绍了 NDIS 小端口驱动、协议驱动、辅助功能驱动等，这些驱动模块基本上与 OSI 模型相对应。

除了基础网络协议栈，Windows 中还包括了一些辅助框架，例如 WSK 框架、WFP 框架、WinPcap 框架等，这些也都是比较常见和常用的网络驱动框架。

第 18 章 Windows 文件系统

文件系统是操作系统用于管理存储介质上文件和目录的方法集和数据结构集，为 Windows API 提供了基于目录机制、对象机制的文件管理方法，这些方法不需要直面刻板晦涩的磁盘系统和二进制数据。无论是 Windows 还是 Linux，都具有相应的文件系统。

本章我们将按照图 18-1 所示的提纲介绍 Windows 文件系统。

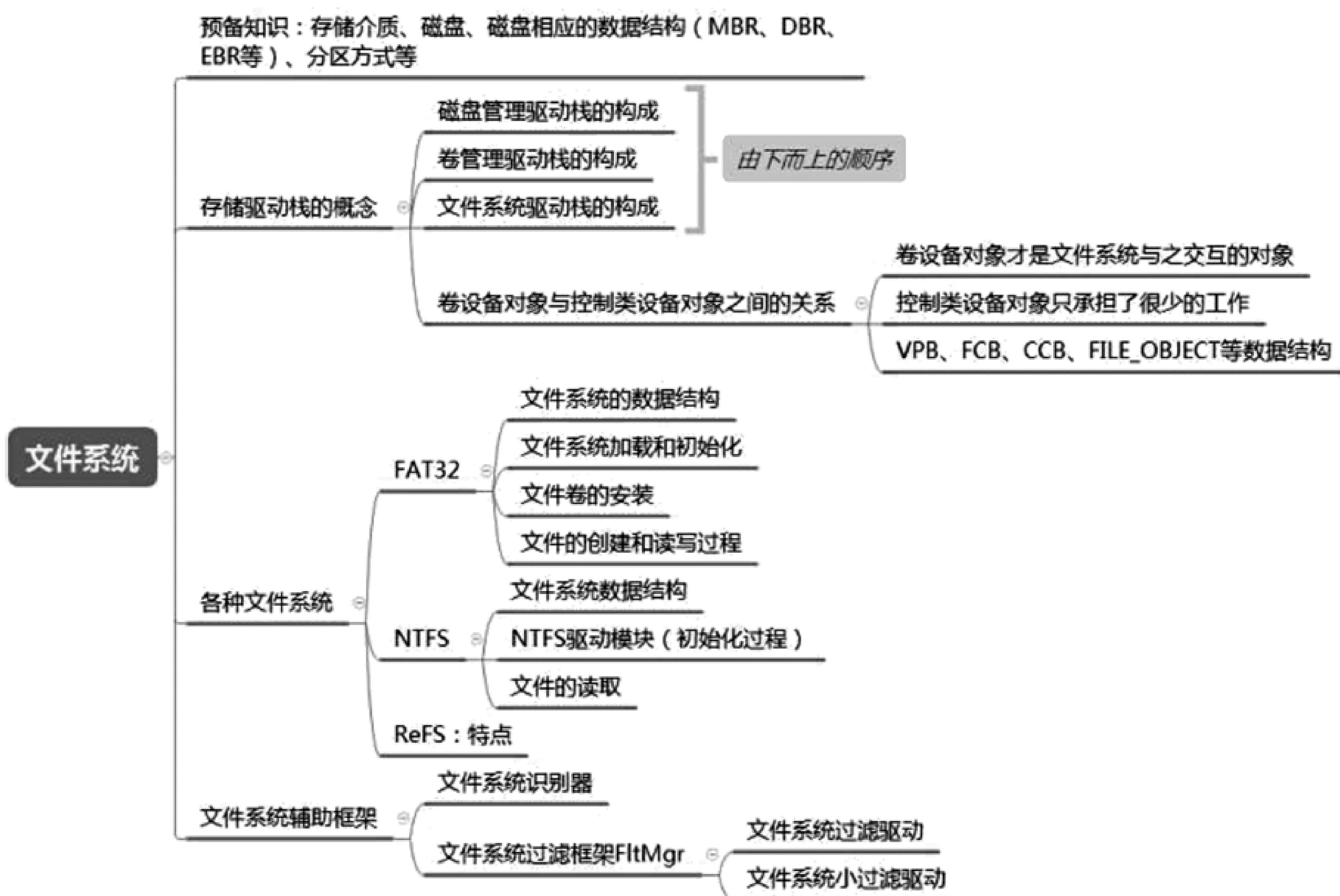


图 18-1　本章提纲

在介绍 Windows 文件系统之前，我们先来看看磁盘的相关知识。

物理磁盘：硬盘或其他种类的磁盘实体。

逻辑磁盘：所谓逻辑磁盘，就是我们平时所说的 D 盘、E 盘等，这是将磁盘扩展分区再次分区后形成的逻辑区域，逻辑磁盘也称为“文件卷”。

磁盘分区：物理磁盘分区有三种类型，即主磁盘分区、扩展磁盘分区、逻辑分区，其中逻辑分区是在扩展分区中划分的。

- 主磁盘分区：即磁盘的启动分区，也是磁盘的第一个分区。计算机中的主磁盘分区至少有 1 个，最多有 4 个，这类分区用于安装操作系统且是不能二次分区的。主磁盘分区是直接从物理磁盘上划分的。
- 扩展磁盘分区：扩展分区不能直接使用，必须进行二次划分逻辑分区才可以使用。计

算机中的扩展分区最多有 1 个,也可以没有,但主分区加扩展分区的总数不能超过 4 个。

- 逻辑分区:是在扩展磁盘分区中划分出来的分区,可以有若干个,逻辑分区也称为“逻辑卷”。

无论怎样,我们在给新磁盘建立分区时都要遵循以下顺序:建立主磁盘分区→建立扩展磁盘分区→建立逻辑分区→激活主磁盘分区→格式化所有分区。

对于磁盘来说,整个磁盘是由若干分区构成的,每个分区由若干磁道构成,而每个磁道又是由许多扇区组成,每个扇区的大小是 512 B,如图 18－2 中左边所示。

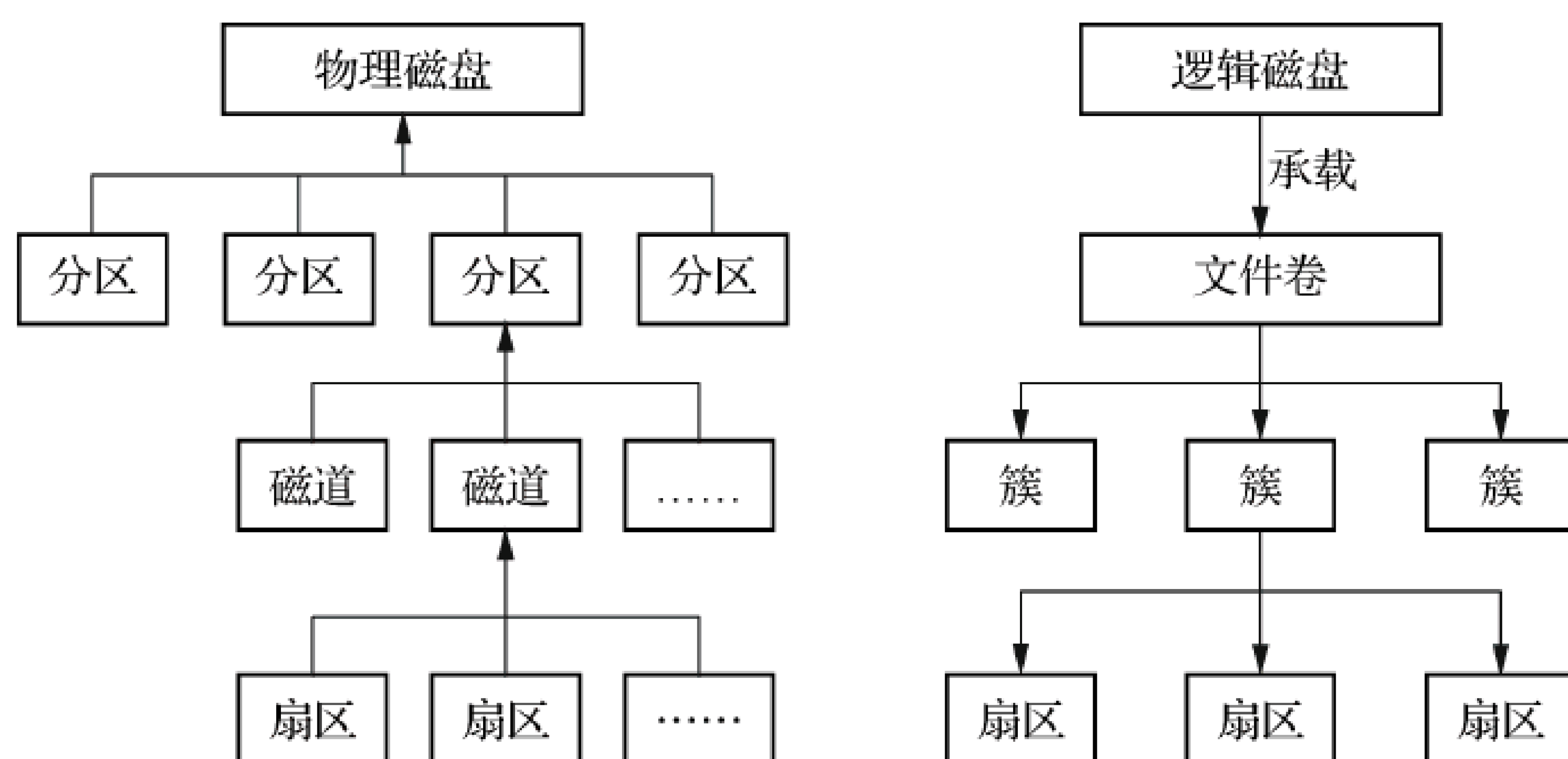


图 18－2 物理系统中的物理磁盘和 FAT 文件系统中的逻辑磁盘的构成

而从文件系统的角度来看,在 Windows 系统中逻辑磁盘(这个逻辑磁盘可以代表一个物理磁盘,也可以代表物理磁盘的一个分区)是承载文件卷(逻辑卷)的载体,逻辑卷又可以分成若干个簇,每个簇是由若干扇区组成的,如图 18－2 中右边所示。在 32 位系统中,扇区的大小是 512B,扇区是文件系统访问磁盘的最小单位。

磁盘按照接口类型一般可分为 IDE 硬盘、SATA 硬盘、SCSI 硬盘和 SAS 硬盘。IDE 硬盘的价格低,性能好,应用非常广泛,不过当前 SATA 硬盘已经逐渐取代了 IDE 硬盘成为 PC 的主流硬盘。SCSI 硬盘则由于更好的性能普遍应用于服务器系统中。SAS 是串行连接 SCSI,是新一代的 SCSI 技术,具有更高的传输速度,因而 SAS 硬盘比 SCSI 硬盘的速度更快。

MBR(Main Boot Record):主引导记录,占 446 字节,是计算机启动后从可启动介质上首先载入内存并且执行的代码,通常用来解释磁盘分区结构。

DBR(DOS Boot Record):DOS 引导记录,是由硬盘的 MBR 装载的程序段,通常用来解释文件系统,存放于硬盘分区后每个分区的第一个扇区。DBR 载入内存后,随即开始执行该引导程序段,其主要功能是完成操作系统的自举并将控制权交给操作系统。硬盘的每个分区都有引导扇区,但只有被设置为活动分区的才会被 DBR 载入内存运行。

EBR(Extended Boot Record):扩展引导记录,类似于 MBR。因为 MBR 的 4 条分区信息的限制,可以使用 EBR 方便扩展分区数量。EBR 的结构与 MBR 类似,但是没有引导程序和磁盘签名,仅仅保留了分区表和结束标志。

我们以 4 个分区的硬盘为例,按照在磁盘中位置由低到高的顺序依次为:主引导扇区→

基本分区→扩展分区。

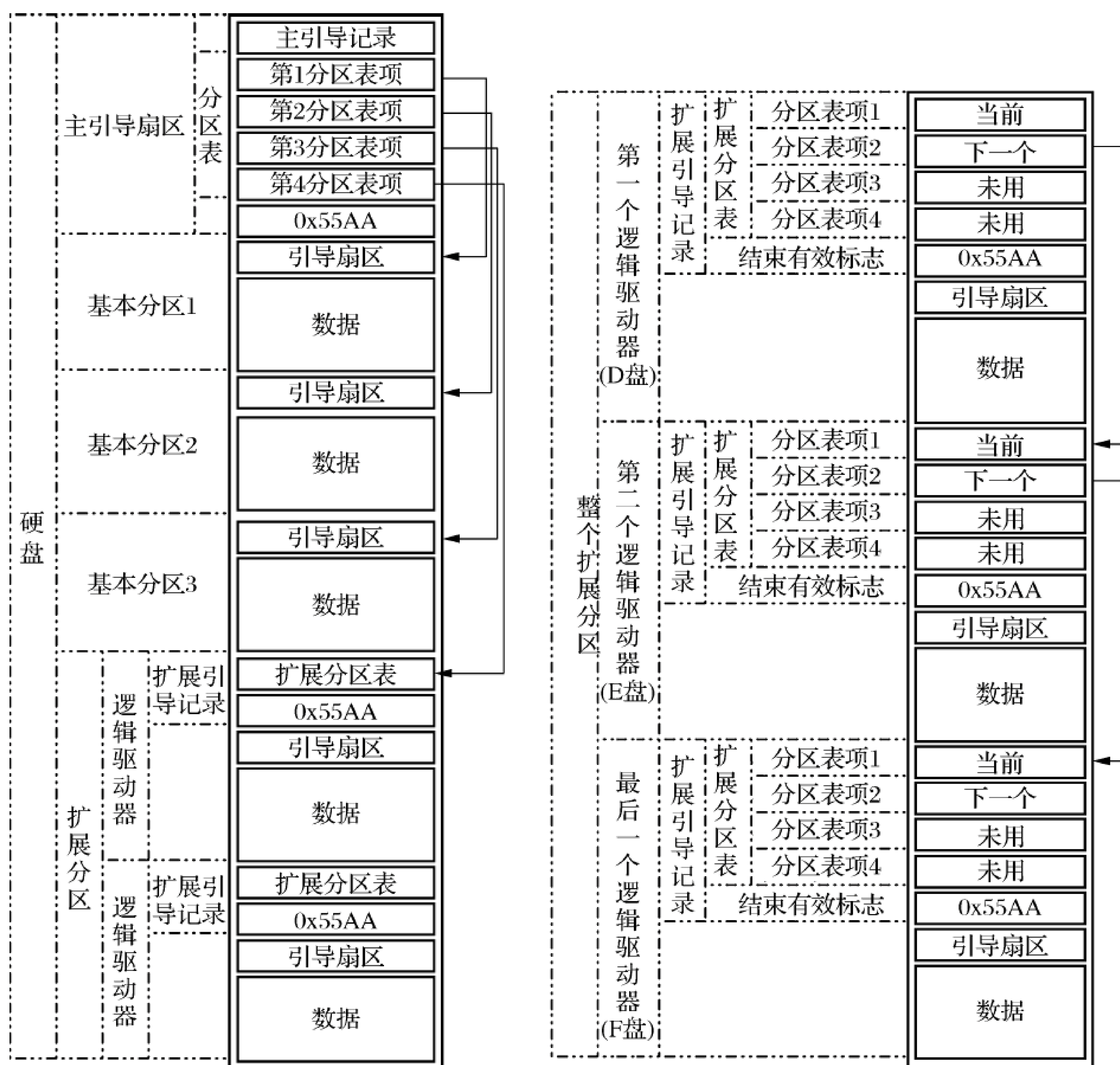


图 18－3　MBR 结构下硬盘与其扩展分区的结构(图片来自 CSDN)

主引导扇区:共占用 512 B 的空间(正好一个扇区),如图 18－3 的左半部分示,包括:

- 主引导记录(446 B)。
- 硬盘分区表(Disk Partition Table,DPT),包含了 4 个分区表项,共占用 64 B 的空间(16 B×4)。其中 4 个硬盘分区表项分别指向了主引导记录之后的基本分区 1～3 的引导扇区和扩展分区的第一个扩展分区表。DPT 只有 64 B,也就是说最多只能描述 4 个分区。这也是主分区和扩展分区不能超过 4 个的原因。
- 分区有效标志 0x55AA(2 B)。

基本分区:每个基本分区都包含了引导扇区和存储的数据两部分内容。

扩展分区:可以分为多个逻辑驱动器(例如 D 盘、E 盘等),如图 18－4 所示,其中的每个逻辑驱动器都有一个类似于 MBR 的扩展引导记录(EBR),EBR 包括一个扩展分区表和该扇区的结束标志(0x55AA),当然如果磁盘上没有扩展分区也就不会有 EBR 和逻辑驱动器。第一个逻辑驱动器的扩展分区表中的分区表项 1 指向它自身的引导扇区。分区表项 2 指向

下一个逻辑驱动器的 EBR,如果不存在下一个的逻辑驱动器,分区表项 2 就不会使用且被赋值为 0。扩展分区表的分区表项 3 和分区表项 4 都没有意义,不会被使用。

除了扩展分区上最后一个逻辑驱动器外,扩展分区表的格式在每个逻辑驱动器中都是重复的:分区表项 1 标识了逻辑驱动器本身的引导扇区,分区表项 2 标识了下一个逻辑驱动器的 EBR。最后一个逻辑驱动器的扩展分区表只会列出它本身的分区项。



图 18－4　主分区与扩展分区

在 Windows 体系下磁盘分区有以下两种方式:

- MBR 方式:MBR 仅为分区表保留了 64 B 的存储空间,因此最多分为 4 个分区。
- GPT 方式:GPT 磁盘分区结构解决了 MBR 只能分 4 个主分区的缺点,分区数理论上是没有限制的。

GPT(GUID Partition Table):全局唯一标识磁盘分区表,其数据结构如图 18－5 所示:

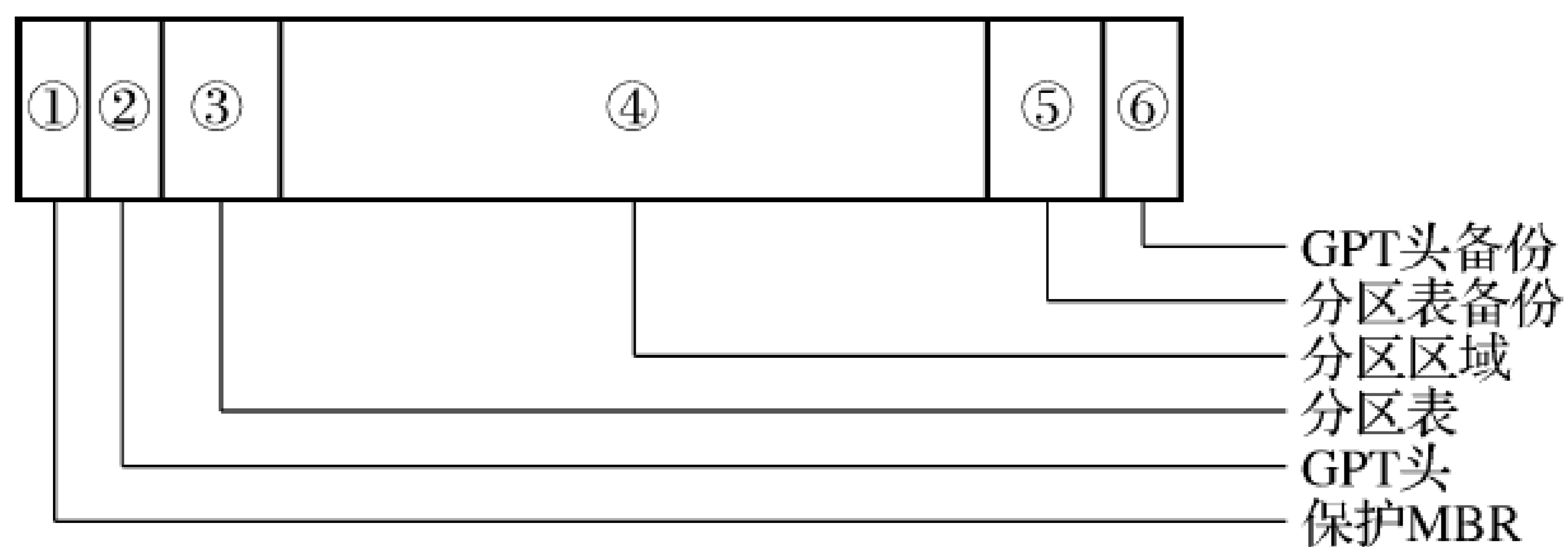


图 18－5　GPT 的结构

- 保护 MBR:位于 GPT 磁盘的第一个扇区,也就是 0 号扇区,由磁盘签名、MBR 磁盘分区表和结束标志组成,没有引导代码,而且分区表内只有一个分区表项,GPT 根本不用这个表项,它只是用来让系统认为这个磁盘是合法的。
- GPT 头:位于 GPT 磁盘的第二个扇区,也就是 1 号扇区,该扇区是在创建 GPT 磁盘时生成的,其作用是定义分区表的位置和大小。GPT 头还包含了头和分区表的校验和,这样就可以及时发现错误。
- 分区表:位于 GPT 磁盘的 2～33 号扇区,一共占用 32 个扇区,能够容纳 128 个分区表项。每个分区表项的大小为 128 B。由于每个分区表项用于管理一个分区,因此允许 GPT 磁盘创建 128 个分区,每个分区表项中记录着分区的起始地址、结束地址、分区类型的 GUID、分区名称、分区属性和分区 GUID。
- 分区区域:GPT 分区区域就是用户使用的分区,也是用户进行数据存储的区域。分区区域的起始地址和结束地址由分区表定义。

➢ GPT 头备份:GPT 头有一个备份,放在 GPT 磁盘的最后一个扇区,但这个 GPT 头备份并非完全备份 GPT 头,某些参数会有不同,复制的时候会根据实际情况修正。
➢ 分区表备份:分区区域结束后就是分区表备份,其地址在 GPT 头备份扇区中有描述。分区表备份是对分区表的 32 个扇区的完整备份。如果分区表被破坏,系统会自动读取分区表备份,以保证正常识别分区。

可以看出,GPT 分区结构相对于 MBR 要简单许多,并且分区表以及 GPT 头都有容灾备份。

18.1 Windows 存储驱动栈

磁盘的最小单位是扇区,在 X86 的 32 位系统下,一个扇区的大小是 512 B。一个磁盘往往被分割成若干分区,每个分区包含了连续的多个扇区。但磁盘一般是以"卷"为单位来管理的,卷分为两种:简单卷和多分区卷。简单卷对应一个独立的分区;多分区卷管理多个分区,构成该卷的多个分区可以是同一磁盘中不连续的分区,甚至是不同磁盘的不同分区。其实可以把卷看作扇区的逻辑集合,这些扇区可以位于一个磁盘或者多个磁盘。但是从文件系统这个层面看过去只能看到卷,却无法识别卷内的扇区分布于哪些磁盘,这也是卷作为文件系统底层结构对于磁盘扇区的逻辑抽象。

图 18－6 是 Windows 存储驱动体系的框架。可以看出,整个存储驱动体系分为三层,自下而上分别是磁盘管理驱动栈、卷管理驱动栈、文件系统\驱动栈。之所以叫"驱动栈"是因为每层都是由多个驱动程序堆叠而成。同时,驱动栈也是设备栈,驱动会生成对应的设备对象并堆叠起来,这与一般的驱动设备堆叠是一样的。文件系统不直接跟磁盘体系打交道,而是要通过卷的中转,其中磁盘管理驱动栈和卷管理驱动栈是在 I/O 系统初始化的时候就构建好了的。

1) 磁盘管理驱动栈

➢ atapi. sys 是磁盘管理驱动栈中的 IDE 总线驱动模块,负责通知 PNP 管理器有新的 IDE 磁盘挂入/卸载。当然不只是磁盘,所有 IDE 接口的设备都是由 atapi. sys 枚举的。如果是 SCSI 磁盘,这个位置会由 scsiport. sys 代替。
➢ acpi. sys 是总线过滤型驱动模块,其过滤设备对象由 atapi. sys 指示创建。
➢ disk. sys 是磁盘类驱动模块,所谓类驱动是相对于小端口驱动而言的,类驱动将磁盘的共性功能抽象出来作为通用模块,而将依赖于具体磁盘型号的个性功能下沉成为小端口驱动。disk. sys 是磁盘的共性功能驱动,负责磁盘通用功能的逻辑实现。
➢ partmgr. sys 是磁盘栈的分区管理模块,用于通知 PNP 管理器当前磁盘上有哪些分区,这样卷管理驱动程序就可以知道要创建或删除哪些分区。

2) 卷管理驱动栈

➢ ftdisk. sys 是卷管理驱动程序,也是卷管理驱动栈中的总线驱动模块,负责通知 PNP

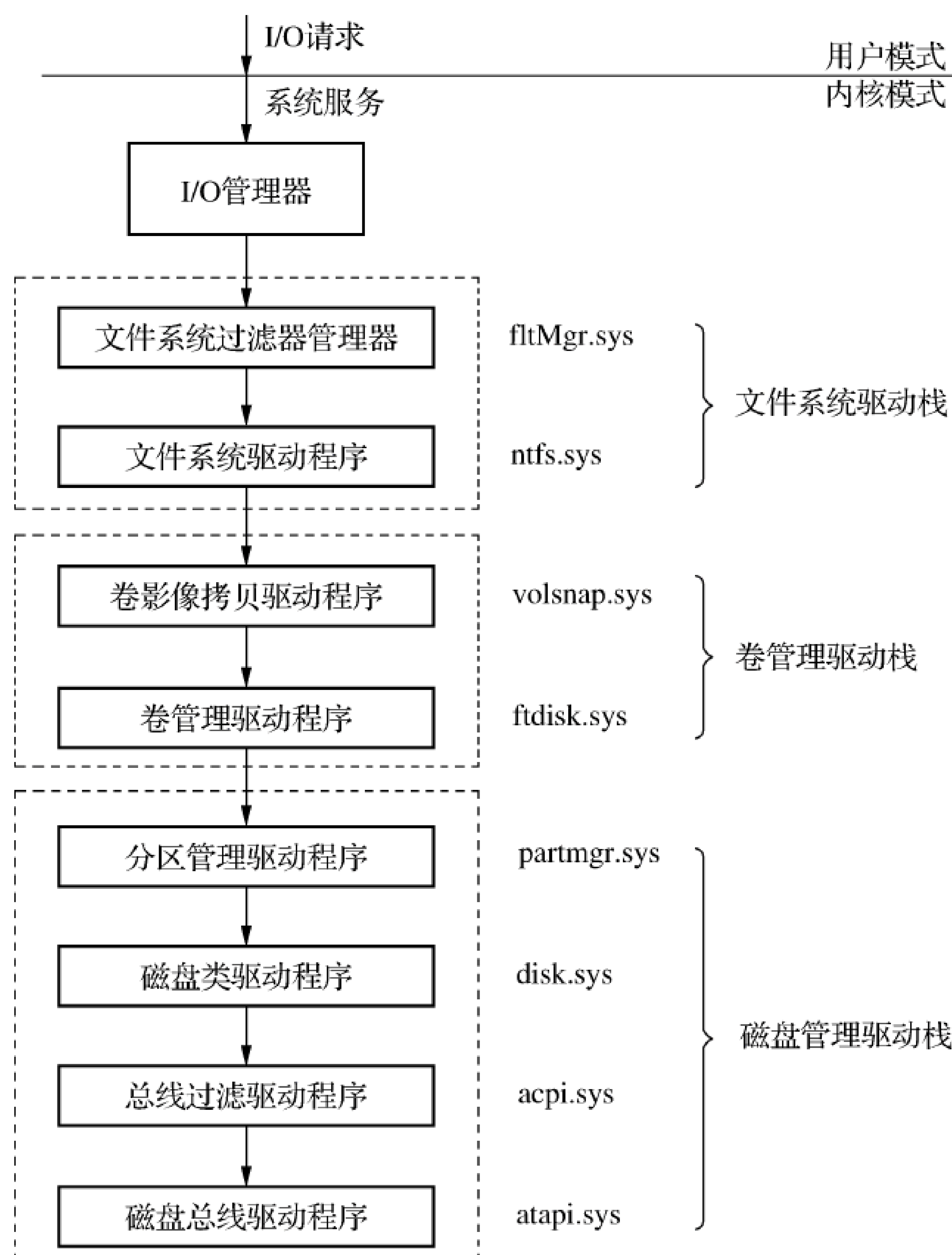


图 18－6　Windows 7 之前的存储驱动体系框架

管理器有哪些卷。当磁盘分区发生变化时，partmgr. sys 指示卷管理驱动创建对应的卷设备对象，在 Windows 7 及以上的版本中这个位置由 volmgr. sys 代替。

➢ volsnap. sys 是卷影像拷贝驱动程序，也是卷管理驱动栈中的功能驱动模块，提供了卷的附加功能。

3）文件系统驱动栈

➢ ntfs. sys 是文件系统驱动栈中具体文件系统的实现，除了 NTFS 文件系统，这个位置也可以是代表 FAT 文件系统的 fastfat. sys。

➢ fltMgr. sys 是文件系统过滤管理器，是整个文件系统驱动栈乃至存储驱动栈最顶层的模块。它提供了一个管理框架，以方便开发人员编写文件系统过滤驱动。

从图 18－7 可以看出，Windows 7 下卷管理驱动栈中的总线驱动模块会生成一个物理设备对象，这是卷管理器上的总线设备。在此之上还生成了个功能设备对象，即 VolMgrControl，这是总线设备上的功能设备对象。在这个管理器功能设备对象之上 volmgr. sys 又生成了 4 个卷设备对象（系统中有 4 个分区），即 HarddiskVolume1、HarddiskVolume2、HarddiskVolume3、HarddiskVolume4，且是物理设备对象，代表了 4 个分区，都是由卷管理器生成的，在这 4 个分

区设备对象之上才是卷的功能设备对象以及更上层的提供了卷附加功能的设备对象。

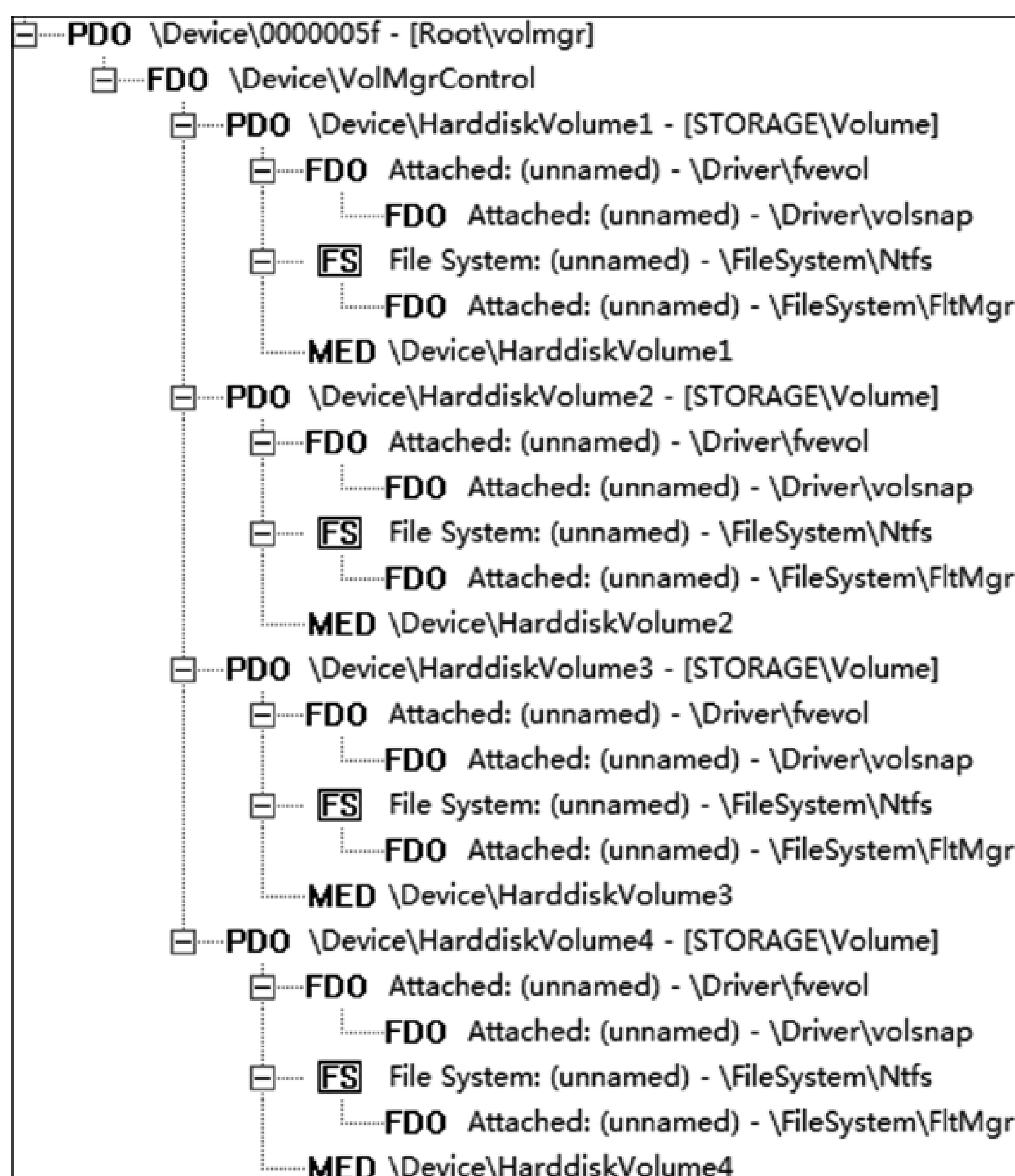


图 18－7　Windows 7 系统下卷以上存储驱动栈的框架

在卷管理驱动栈之上是文件系统驱动栈。作为中间层的卷不但与下层的磁盘驱动发生关联，也要与上层的文件系统产生联系，这个过程叫作“卷识别”。没有文件系统的参与就不可能按目录访问磁盘上的文件。

但是卷识别的过程并不是发生在 I/O 系统初始化的时候，而是发生在上层进程第一次访问某个卷上的目录或文件的时候：当 I/O 管理器或对象管理器解析到卷路径名的时候，就会调用 IopCheckVpbMounted 进而调用 IopMountVolume 方法进行卷识别。

IopMountVolume 首先选择全局文件系统列表（文件系统安装的时候会注册到全局列表中），并对列表中的每个文件系统下发主功能码为 IRP_MJ_FILE_SYSTEM_CONTROL、副功能码为 IRP_MN_MOUNT_VOLUME 的 IRP，这个 IRP 代表了挂载请求，挂载是由文件系统中的功能函数来定义和实现的。如果该 IRP 返回成功，则证明该卷和文件系统可以挂载并且挂载成功，继而完善文件卷参数块（VPB）等数据结构；否则，尝试列表中的下一个文件系统。可以看出，卷识别是个文件系统枚举的过程。

VPB（Volume Parameter Block）是文件系统中的重要数据结构，其内容是由 I/O 管理器定义的，其任务是绑定卷设备/块设备（如磁盘分区或虚拟磁盘）和此卷设备对应的文件系统（如 FastFat、NTFS），可以看作块设备和文件系统的连接器。卷设备是无法被 I/O 管理器直接读写的，虽然卷的读写单位是扇区，但如果通过扇区来访问文件是极为不方便的，更无法使用文件目录，因此位于卷上面的文件系统才是我们创建目录，根据目录访问文件、管理文

件的媒介。VPB 就是这二者之间的纽带，其数据结构如下所示：

```
typedef struct_VPB
{
    CSHORT              Type;     // IO_TYPE_VPB
    CSHORT              Size;     //VPB 结构的大小
    USHORT              Flags;    //标志位，含义如下：
                                  //VPB_MOUNTED:此卷已被文件系统识别并已挂载
                                  //VPB_LOCKED:此卷已被文件系统锁定
                                  //VPB_PERSISTENT:将 VPB 一直保留在内存中(不释放)
                                  //VPB_REMOVE_PENDING:此存储设备即将被卸载/删除
                                  //VPB_RAW_MOUNT:指定此卷仅由系统 RAW 文件系统接管
    USHORT              VolumeLabelLength;   //卷标长度
    struct_DEVICE_OBJECT * DeviceObject;    //指向文件系统驱动生成的设备对象，是在安装文件卷的过
                                            //程中完成指向的
  struct_DEVICE_OBJECT * RealDevice;        //指向块设备(如磁盘等)驱动的设备对象
  ULONG               SerialNumber;         //卷序列号
  ULONG               ReferenceCount;       //VPB 的引用计数，用以控制 VPB 的生命周期
  WCHAR               VolumeLabel[MAXIMUM_VOLUME_LABEL_LENGTH / sizeof(WCHAR)];
                                            //卷标，最长 32 个双字节
} VPB, * PVPB;
```

当然也不是每种存储设备都创建 VPB，需要创建 VPB 的存储设备类型只有 4 种：FILE_DEVICE_DISK、FILE_DEVICE_VIRTUAL_DISK、FILE_DEVICE_CD_ROM 和 FILE_DEVICE_TAPE。从上述设备种类的名称也可以知道这些设备的具体类型含义，它们都是承载文件卷的块设备。

如图 18－8 所示，文件系统安装过程中，在通过 IoCreateDevice 创建设备对象时创建 VPB 数据结构(通过 IopCreateVpb 方法创建 VPB)，设备对象中的 VPB 指针指向该结构，VPB 中的 DeviceObject 指向文件系统卷设备，RealDevice 指向块设备，DeviceObject 是在安装文件卷的时候完成指向的。

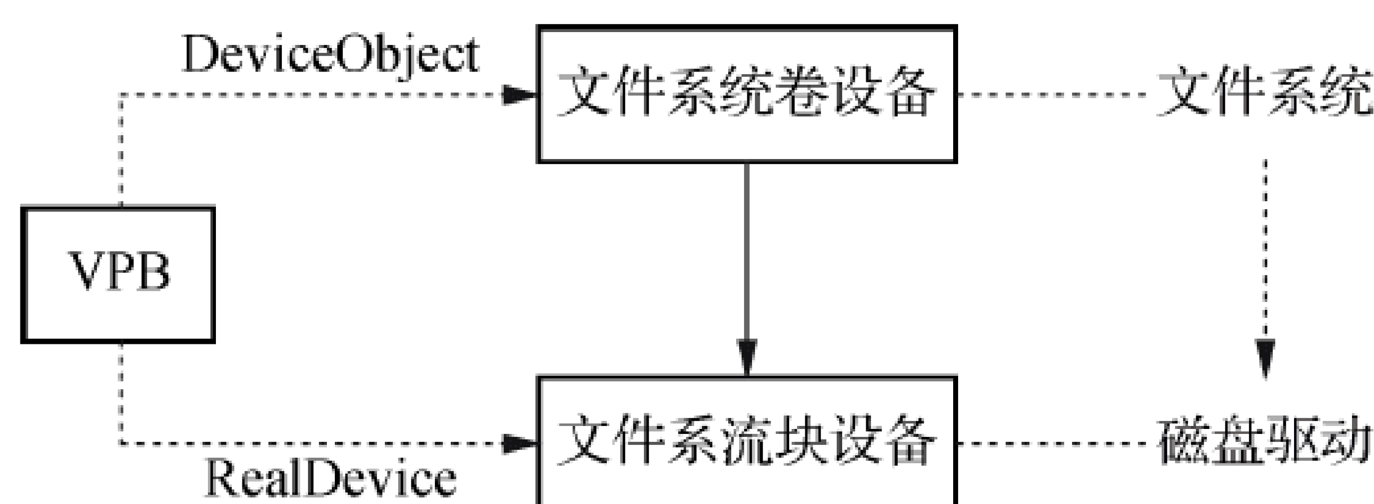


图 18－8　VPB 的连接关系

文件系统一般生成两类设备对象，一类是代表文件系统驱动本身的控制类设备对象(Controller Device Object，CDO)，其任务是修改整个驱动的内部配置；另一类是文件系统卷设备对象，代表了逻辑磁盘，一般一个卷对应一个逻辑磁盘，也就是被这个文件系统挂载(Mount)的卷(Volume)，这类设备对象一般没有设备名，但却堆叠于卷管理器生成的卷设备对象之上，且与 CDO 共享文件系统的功能函数，如图 18－9 所示。

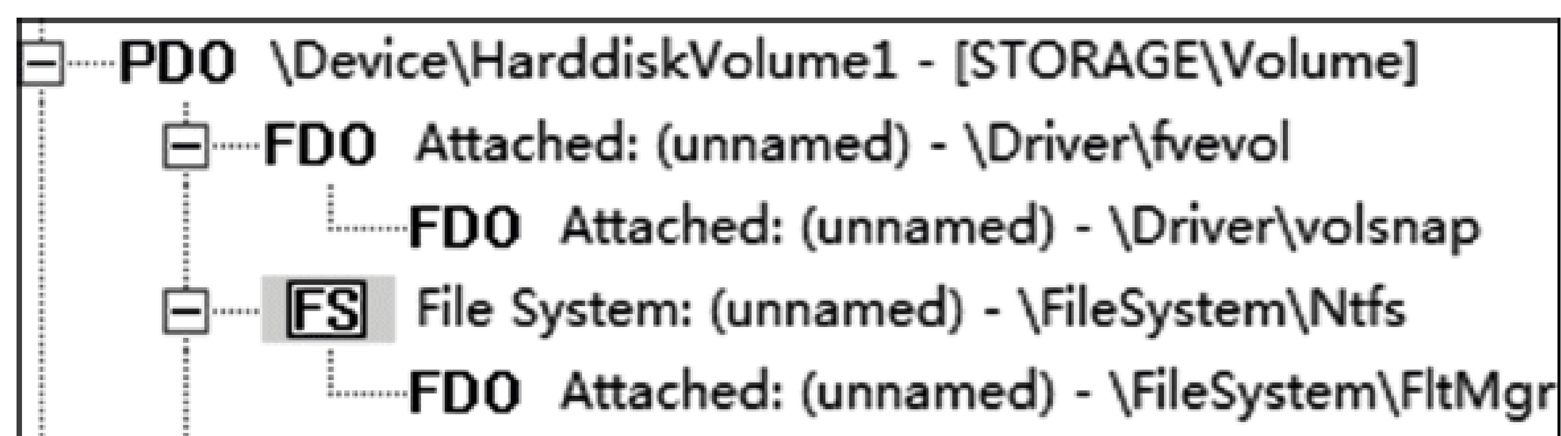


图 18－9　卷管理器与文件系统各自生成的卷设备对象

一个文件系统只对应一个 CDO,但可以对应多个卷设备,甚至可以在一个系统中出现 FAT32 卷设备对象和 NTFS 卷设备对象共存的情况。文件系统在识别出卷的时候会生成一个文件系统卷设备和卷控制块(Volume Control Block,VCB)数据结构,VCB 是由文件系统内部自定义的,通常作为文件系统卷设备对象的扩展区,用于关联卷与文件系统(区别于关联块与文件系统的 VPB)。也可以把 VCB 看作文件系统的全局信息容器。

NTFS 设备对象是由 NTFS 文件系统驱动创建的,这是个 CDO,用于注册到系统全局队列中。同时卷与文件系统挂载后也生成了与卷数量对应的文件系统卷设备对象,如图 18 - 10 中的 unnamed 对象,这些设备对象是未命名的,且与 CDO 一起共享文件系统的功能函数。

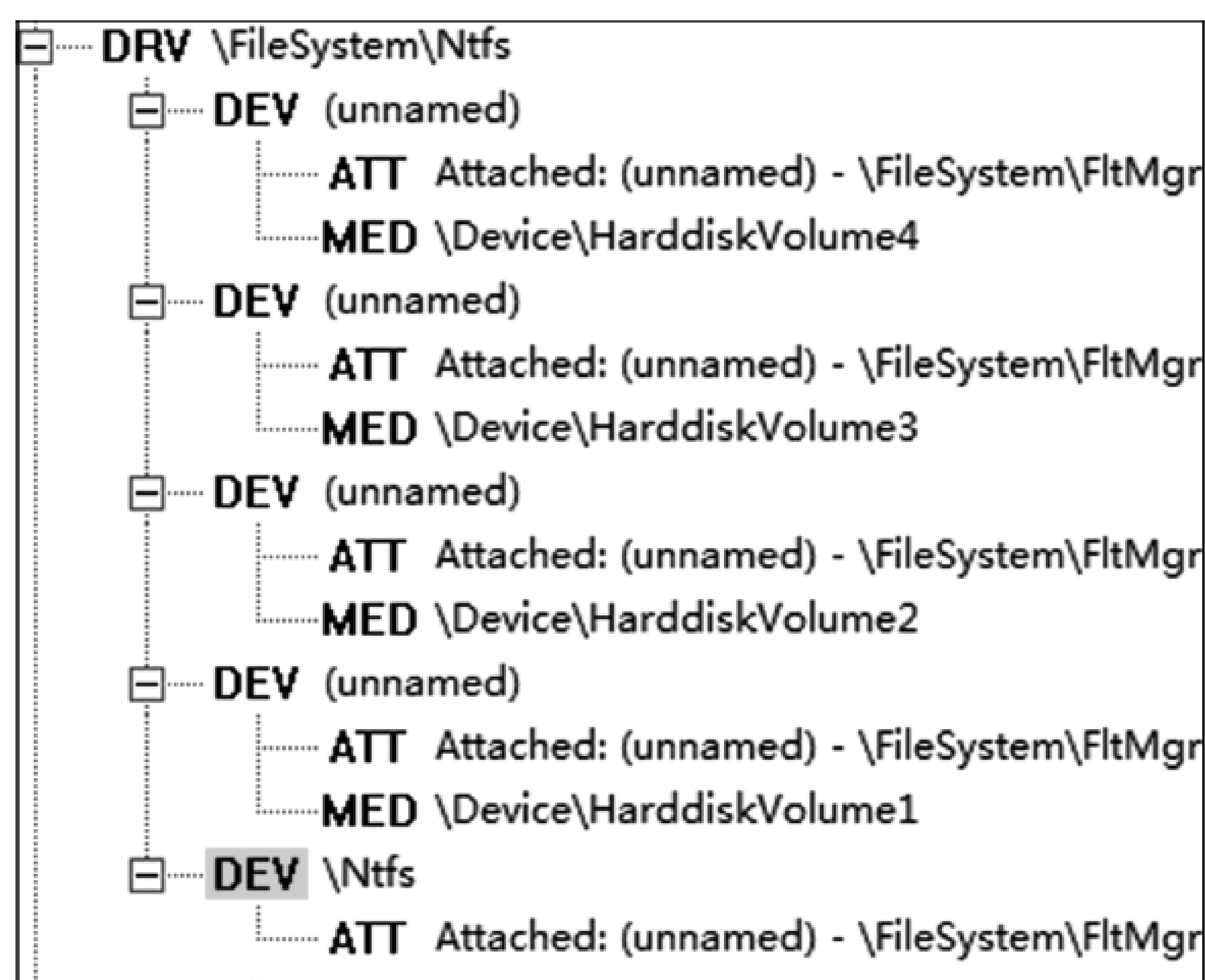


图 18 - 10　CDO 与文件系统卷设备对象

创建文件对象时,上层应用程序将目录 + 文件名通过 I/O 管理器的 IopCreateFile 传递到对象管理器的相应方法中进行解析。当解析到卷设备对象的符号链接时(例如“\C:”)就重定位到卷设备对象中(例如“\Device\HarddiskVolume1”),继而调用卷设备对象的解析函数 IopParseDevice 来解析目录中卷设备符号链接之后的内容。这时就要检查卷是否已被识别,若未被识别,就先下发 MOUNT 的 IRP 进行挂载(识别),挂载成功后再进行后续的解析;已经被识别就根据卷设备对象中的 VPB 找到文件系统设备对象,并从这个设备对象上溯到文件系统设备栈的最顶层设备对象,之后下发 IRP(IRP_MJ_CREATE)给这个最顶层设备对象,以调用对应文件系统的文件创建功能函数来创建文件。

文件系统一般以文件控制块(File Control Block,FCB)来代表对文件或目录的打开,这里的“打开”是全局视野,而不是单个进程对文件的打开,FCB 中包含了文件或目录的基本属性,是文件打开操作的上下文。FCB 在全局中只有一个,只要被打开就创建 FCB,无论有多少进程线程来引用/操作这个文件,FCB 都只有一个。FCB 数据结构的各个成员变量是由文件系统驱动定义的,因此 NTFS 和 FAT32 文件系统中的 FCB 是不一样的,毕竟文件的打开是要通过文件系统的,文件系统也理应对于打开的上下文拥有最终解释权。

文件系统一般以上下文控制块(Context Control Block,CCB)来代表上层应用对文件的每一次打开。因此对于一个已打开的文件,FCB 只有一个,CCB 却可能有多个。两个进程对于同一文件的访问会产生两个不同的 CCB。CCB 中有指针指向 FCB,且上述两个 CCB 都会指向同一个 FCB。CCB 数据结构的各个成员变量也是由文件系统驱动来定义的,因此不同的文件系统中 CCB 也是不一样的。不过无论是 FCB 还是 CCB,对于文件系统的使用进程(应用程序或 I/O 管理器)都是透明的。应用程序无法直接操纵 FCB 或 CCB。

Windows 内核中一般以 FILE_OBJECT 这个数据结构来代表对文件的打开,这是个"一手托两家"的角色,并且要与文件系统中的打开区别开来。FILE_OBJECT 中有两个重要的域:FsContext 和 FsContext2,其中 FsContext 指向 FCB 或 VCB(文件对象是磁盘上的文件时指向前者,文件对象是文件卷时指向后者);FsContext2 指向 CCB,当文件系统执行 IRP_MJ_CREATE 操作时会设置这两个域值。这几个域之间的关系如图 18 - 11 所示。

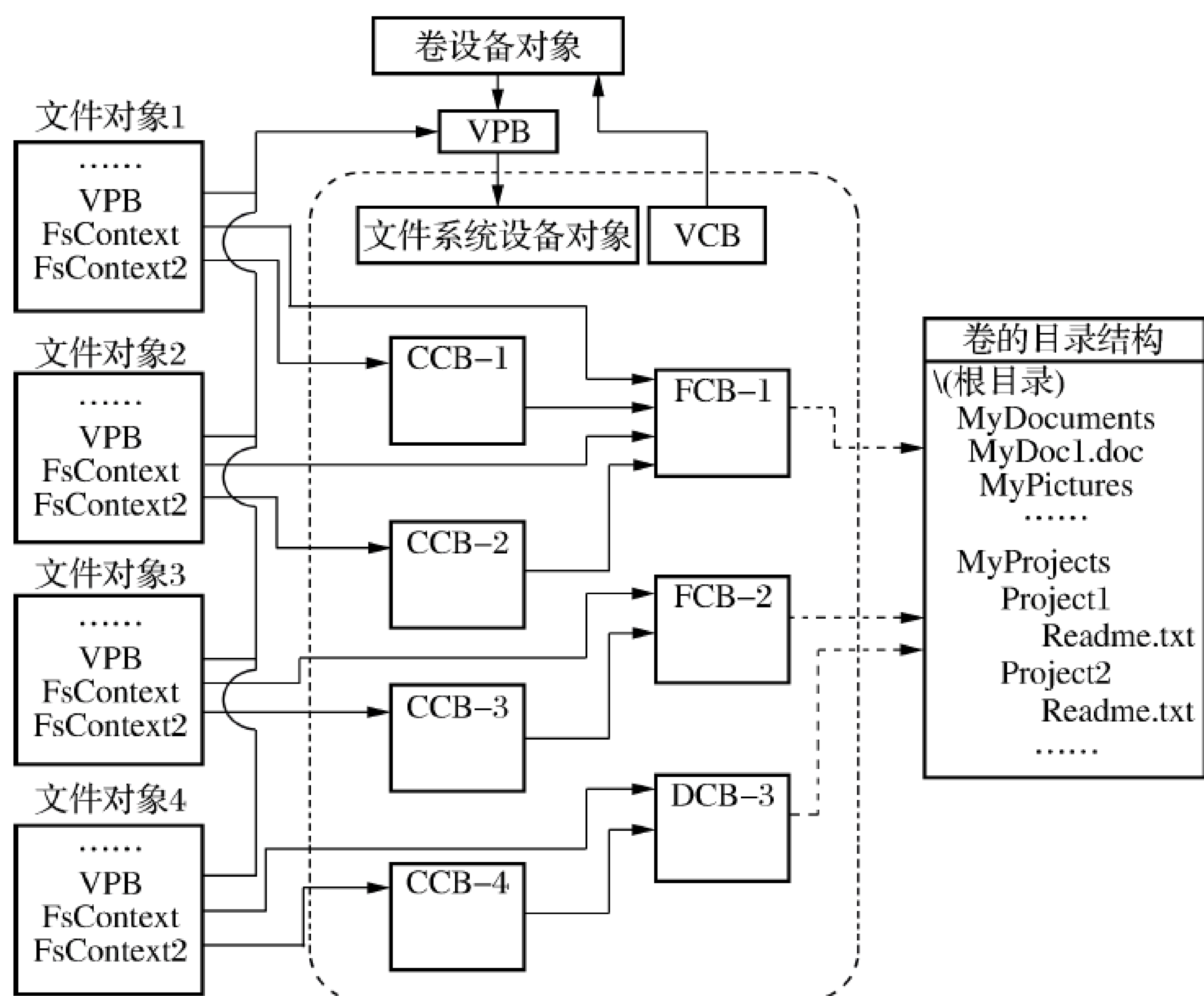


图 18 - 11　FILE_OBJECT 与 FCB、CCB 之间的关系

当执行完系统调用 NtCreateFile 时,也就意味着执行完了 IRP_MJ_CREATE 操作,FILE_OBJECT 分别指向了正确的 FCB/VCB 和 CCB,当对文件进行读写等操作时就可以方便地找到 FCB 和 CCB 了。FILE_OBJECT 中还包括了 VPB 指针,指向 VPB 数据结构,从 VPB 中可以很方便地找到文件系统设备对象和卷设备对象。

图 18 - 11 中,文件对象 1 和文件对象 2 打开的是同一个文件,因此 FCB 相同,但是 CCB 却有两个,这代表了两次不同的打开,而文件对象 3 和文件对象 4 则分别代表了对不同文件的一次打开。

18.2　FAT32 文件系统

FAT32 文件系统是 Windows 中比较古老的一种文件系统,也叫文件系统驱动(File System Driver,FSD),这是一种原理相对简单的文件系统。FAT 指文件分配表(File Allocation Table)。FAT32 也可以说是一套跨平台的文件系统方案,兼容 Windows 和 Linux 两套操作系统。在 Windows 7/Windows 8 大行其道的今天,FAT32 的市场份额已经被 NTFS 远远超过,只是在 U 盘、SD 卡等小容量存储介质中还有较多应用。在 FAT32 之前还有 FAT12 和 FAT16 文件系统(FAT12、FAT16、FAT32 这三种文件系统之间的主要区别在于 FAT 项的大小不同),它们是以 16 位操作系统为基础设计的文件系统,现在已几乎绝迹;在 FAT32 之后还有 exFAT 文件系统(也叫 FAT64),是微软专为闪存(U 盘、存储卡)设备设计的文件系统,兼容性非常好,主要是为了解决 FAT32 不支持 4 GB 以上大文件管理的问题,exFAT 支持的最大文件大小为 16 EB。

FAT32 文件系统本质上是一个以簇号为下标的大数组,数组中存储的信息是该文件下一个簇的簇号,因此一个文件就可以看作一个“簇链”。FAT32 文件系统的映像载体是 fastfat. sys,其依赖的模块只有 ntoskrnl. exe,也就是说 fastfat. sys 依赖的是 Windows 内核,如图 18－12 所示。

fastfat.sys

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntoskrnl.exe	228	00032C08	00000000	00000000	00034850	00004000

图 18－12　fastfat. sys 的依赖模块

1. FAT32 文件系统的数据结构

FAT32 文件系统是以簇为单位分配存储空间的,也是以簇为单位访问文件和目录的。簇由扇区组成,每个簇只能包含 2 的整数次幂数量的扇区,每个簇的大小最大为 64 KB,由于一个扇区的大小为 512 B,因此一个簇最多包含 128 个扇区。文件和目录的内容都存储在簇中,如果一个文件或目录需要多于一个簇的空间,则用 FAT 表来描述和串联多个簇。本质上 FAT 结构用于指出下一簇,同时也说明了簇的分配状态。

在 32 位系统中 FAT 表是以 4 B 为单位进行划分的,每个单元(单位)存储一个簇地址。0 号地址与 1 号地址被系统保留用于存储特殊标志内容。从 2 号地址开始,第 X 号地址对应数据区中的 X 号簇。我们称 FAT 表中的地址为 FAT 表项,FAT 表中记录的值就是 FAT 表项值。

FAT 表项记录着它所代表的簇的有关信息,例如是否为空,是不是坏簇,是否已经是某个文件的尾簇等。对于 FAT32 文件系统来说,FAT 表有两个重要作用:描述簇的分配状态以

及标明文件或目录的下一簇的簇号。

FAT32 文件系统中簇号为 28 位,因此可以覆盖 2^{28} 个 64 KB 簇大小的磁盘空间,因此 FAT32 管理的最大空间为 2 TB。FAT32 文件系统的核心是簇描述项,这是一个 4 B 大小的数据结构,项值的含义是这样的:

- **0X00000000**:表示本簇空闲;
- **0X00000001 ~ 0X0FFFFFF7**:表示同一个文件中当前簇的下一个簇的簇号,其中 0X00000001 和 0X0FFFFFF0 ~ 0X0FFFFFF6 被保留使用;
- **0XFFFFFFFF**:表示本簇是所在文件中的最后一个簇。

目录是 FAT32 文件系统中一种特殊的文件,FAT32 文件系统的每一个文件和文件夹都被分配到一个目录项中,目录项中记录着文件名、文件大小、文件内容起始地址以及其他一些元数据。FATDirEntry 是 FAT32 文件系统中的**目录项**结构,这是个 32 B 大小的数据结构,存储了文件的各种信息,其数据结构是这样的:

```
typedef struct_FATDirEntry
{
    union
    {
      struct {
              unsigned char Filename[8],   //如果 0 号元素为 0XE5,表示这是一个被删除的文件
              Ext[3];
              };//这里要注意的是,文件名只占用 11 个字节,如果文件名很长,就要再分配若干个_slot 结
                //构(每个这样的结构有 13 个字节的空间可用)以承载文件名
      unsigned char ShortName[11];
    };
    unsigned char  Attrib;
    unsigned char  lCase;
    unsigned char  CreationTimeMs;
    unsigned short CreationTime,CreationDate,AccessDate;
    union
    {
      unsigned short FirstClusterHigh;    //用于 FAT32,代表了文件中第一个簇的簇号的高位部分
      unsigned short ExtendedAttributes;  //用于 FAT12/FAT16
    };
    unsigned short UpdateTime;
    unsigned short UpdateDate;
    unsigned short FirstCluster;   //代表了文件中第一个簇的簇号的低位部分
    unsigned long  FileSize;       //这是个 4 B 的域值,因此 FAT32 表示的最大文件是 4 GB 的
}FATDirEntry, * PFATDirEntry;
```

在 FAT32 文件系统下,磁盘物理分为 4 部分:

- **保留区 1(含 MBR)**:占用一个扇区大小。
- **保留区 2(含 DBR)**:FAT32 系统下的保留区 2 占用多个扇区(比较常见的为 32、34 或 38 个扇区),除了 DBR,还包括一个 FSINFO 信息扇区和 DBR 备份扇区。FSINFO 信息扇区用于记录文件系统中空闲簇的数量以及下一可用簇的簇号等信息,以供操作系统参考。FSINFO 扇区一般位于文件系统的 1 号扇区,结构也非常简单。
- **FAT 区**:由 FAT1 和 FAT2 构成,FAT2 紧跟在 FAT1 后面,其位置可以通过 FAT1 的位置加上 FAT1 的扇区数计算出来,是 FAT1 的备份。FAT 实际上是个固定长度的数组,文件卷上的每一簇在 FAT 中都有对应的一项。

➢ **数据区**：数据区是用于保存数据的，包括根目录也保存于此，这一区域以簇而不是扇区为管理单位。

这里要注意的是，在 FAT32 文件系统中，FAT1 与 DBR 不是紧挨着的，而在 FAT16 文件系统中它们则是相邻的，如图 18－13 所示。

当 FAT 文件系统被创建时，FAT 表会被清空，在 FAT1 和 FAT2 表中的 0 号与 1 号地址被写入特定值。由于创建文件系统的同时会创建根目录，因此也要在数据区为根目录分配一个簇的空间（2 号簇作为起始簇）。





图 18－13　FAT16 与 FAT32 文件系统的布局对比

在 FAT 文件系统中，文件系统的元数据记录在引导扇区中。引导扇区位于整个文件系统的 0 号扇区，是文件系统隐藏区域（也称为保留区）的一部分，我们称其为 DBR（DOS 引导记录）扇区，DBR 中记录着文件系统的起始位置、大小、FAT 表个数及大小等相关信息，如图 18－14 所示。

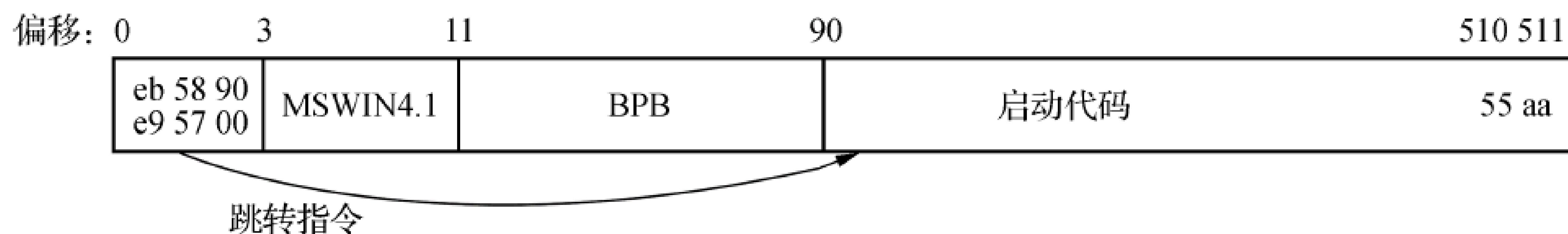


图 18－14　FAT32 文件系统的 DBR 结构

从图 18－14 可以看出，DBR 包含了：

➢ 用于跳转到启动代码的跳转指令，共 3 B；

➢ 厂商标志（MS，代表 Microsoft）和操作系统版本号（WIN4.1），共 8 B；

➢ 基础输入输出系统参数块（BIOS Parameter Block，BPB），共 79 B；

➢ 启动代码，共 420 B；

➢ 结束标志符，共 2 B。

在 FAT 文件系统中，同时使用扇区地址和簇地址两种地址管理方式。只有存储用户数据的数据区使用簇进行管理（FAT12 和 FAT16 文件系统的根目录除外），所有簇都位于数据区。其他文件系统管理数据区域时不是以簇地址进行管理的，而是使用扇区地址进行管理。文件系统的起始扇区为 0 号扇区（逻辑 0 扇区）。

虽然原则上 FAT32 文件系统允许根目录位于数据区的任何位置，但通常情况下它都位

于数据区的起始扇区，数据区起始扇区号即根目录扇区号。在 FAT 文件系统中，先要寻找数据区的第一簇（即 2 号簇）的地址，它不是位于文件系统开始处，而是位于数据区。在数据区前面是保留区域和 FAT 区域，再往前还有 MBR 区域，这些区域都不使用 FAT 表进行管理。因此，数据区以前的区域只能使用扇区地址而无法使用簇地址。

2. FAT32 文件系统的加载和初始化

fastfat.sys 很显然是个驱动模块（NT 式驱动），其入口函数 DriverEntry 的主要功能包括：

- 创建设备对象 Fat，这是一个 **FILE_DEVICE_DISK_FILE_SYSTEM** 类型的设备对象，也是一个 CDO，但不是指块设备。所谓块设备就是磁盘这样的设备，块设备对象是由磁盘驱动创建的。Fat 这个设备对象是文件系统驱动创建的控制设备对象，用于修改配置、挂入全局文件系统队列。
- 初始化驱动对象的功能函数为 VfatBuildRequest，FAT32 文件系统的驱动程序支持 IRP_MJ_READ\IRP_MJ_WRITE 等较为常规的主功能码。
- 初始化三个私有列表：**FcbLookasideList**、**CcbLookasideList** 和 **IrpContextLookasideList**，并设置快速 I/O 函数。这几个成员变量都存在于设备对象 Fat 的扩展区域里，这个扩展区域被定义为 VFAT_GLOBAL_DATA 类型。
- 调用方法 IoRegisterFileSystem 注册 FAT32 文件系统。全局系统中有 4 类文件系统列表（后两种类型的文件系统现在已经很少用了），分别是：
  - 磁盘文件系统驱动列表 **IopDiskFsListHead**：存放的文件系统设备对象类型为 FILE_DEVICE_DISK_FILE_SYSTEM。
  - 网络文件系统驱动列表 **IopNetworkFsListHead**：存放的文件系统设备对象类型为 FILE_DEVICE_NETWORK_FILE_SYSTEM。
  - 光盘文件系统驱动列表 **IopCdRomFsListHead**：存放的文件系统设备对象类型为 FILE_DEVICE_CD_ROM_FILE_SYSTEM。
  - 磁带文件系统驱动列表 **IopTapeFsListHead**：存放 FILE_DEVICE_TAPE_FILE_SYSTEM 类型的文件系统设备对象。

IoRegisterFileSystem 就是根据优先级将创建的设备对象 Fat 挂入某个队列的队头或队尾，之后调用方法 IopNotifyFileSystemChange 通知文件系统有变化（新增）。在全局的 FsChangeNotifyListHead 列表中存放着各软件预先注册的数据结构，一旦有文件系统加载或卸载，系统就会遍历这个列表向注册的软件发出通知。

fastfat.sys 没有定义 AddDevice 函数，执行完 DriverEntry 后 I/O 管理器不会调用 AddDevice 方法，因此 Fat 也不会被纳入设备对象堆栈体系。

3. 文件卷的安装

文件卷是建立在硬盘和分区之上的一个逻辑层。块存储设备在安装文件卷之前只是一个原始的存储设备，且只能通过扇区读写，此时并没有一个文件系统来组织文件，更没有文件目录的概念，因此也就无法通过目录管理文件。安装文件卷之后，某种文件系统（例如





FAT32/NTFS)将文件卷组织了起来,可以做到以文件为单位进行文件管理,也可以采用目录方式对文件寻址。文件卷的安装是从磁盘访问升级到文件访问的关键一步。

文件卷的安装是由文件系统完成的。I/O 管理器发出一个要求安装文件卷的 IRP,其主功能码为 IRP_MJ_FILE_SYSTEM_CONTROL,副功能码为 IRP_MN_MOUNT_VOLUME,I/O 管理器调用 IoCallDriver 方法将 IRP 下发到 FAT32 文件系统驱动中,其设备对象是在 DriverEntry 中创建的设备控制对象 Fat。其实在 FAT32 文件系统中,控制设备对象也用来识别 FAT 格式的文件卷。

FAT32 文件系统安装文件卷的函数调用过程如图 18－15 所示。

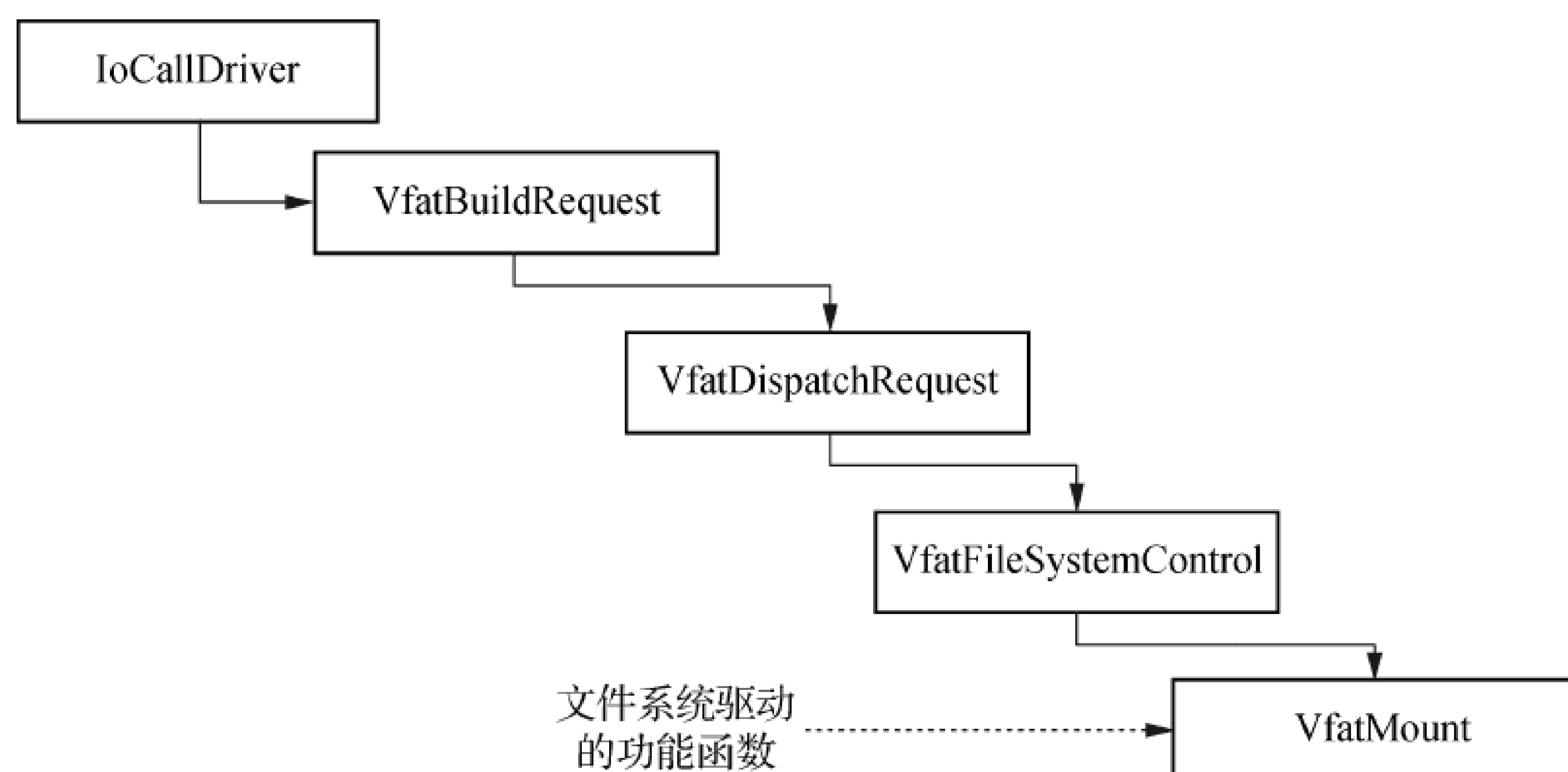


图 18－15　安装文件卷的 IRP 的流向

VfatMount 是文件卷安装的具体执行逻辑,会首先辨认目标设备上的文件是否为 FAT 文件卷,其具体执行步骤如下:

(1) **获取磁盘几何信息**:所谓几何信息,就是磁盘是固定的还是可拆卸的、扇区字节数等基础信息。这是通过向磁盘驱动程序下发主副功能码分别为 IRP_MJ_DEVICE_CONTROL、IOCTL_DISK_GET_DRIVER_GEOMETRY 的 IRP 实现的。

(2) **获取磁盘分区信息**:这是通过向磁盘驱动下发主功能码为 IRP_MJ_DEVICE_CONTROL、副功能码为 IOCTL_DISK_GET_PARTITION_INFO 的 IRP 实现的。获取分区信息后判断是否合法,如果不合法则证明该卷不可辨认,退出 FAT32 文件系统的安装过程。

(3) **读取第一个扇区**:使用 VfatReadDisk 方法获取引导扇区(第一个扇区)中的数据,也就是磁盘引导块,VfatReadDisk 下发 IRP_MJ_READ 的 IRP 到磁盘驱动中读取扇区内容,以验证该卷是否确实为 FAT 文件卷。

(4) **创建文件系统卷设备对象**:这是个类型为 FILE_DEVICE_FILE_SYSTEM 的未命名设备对象,与 CDO(DriverEntry 中创建的 FILE_DEVICE_DISK_FILE_SYSTEM 类型的设备对象)共享 FAT32 文件系统驱动的功能函数,代表了具体的文件卷。这个设备对象的扩展结构就是 VCB。创建文件系统卷设备对象的方法依然是 IoCreateDevice。

(5) **安装磁盘,并获取相关信息记录在 FATINFO 数据结构中**:这是通过 FAT32 文件系

统的 VfatMountDevice 方法实现的。

(6) **创建 VCB 中的 FATFileObject 并初始化**:FATFileObject(FILE_OBJECT)中有 FsContext 和 FsContext2 两个域分别指向 FCB 和 CCB,这里要创建一个名为"\$$FAT$$"的 FCB 和暂存了状态信息的 CCB,并完成指向。

(7) **初始化缓存文件内容的机制**:通过 CcInitializeCacheMap 方法实现。

(8) **将卷标读到 VPB 中**:通过 ReadVolumeLabel 方法实现卷标读取操作。

注意:磁盘的 1 号簇用于存储 FAT 表,FAT 表有多大这个簇就有多大,且 1 号簇没有后继簇。

经过了这些繁琐的步骤后安装就完成了,文件系统为每个卷设备生成了一个未命名设备对象,这些未命名的设备对象就是文件系统的卷设备对象,是文件卷安装后的结果。当使用 NtCreateFile 函数打开某个目录/文件时,遇到卷标后转到解析函数 IopParseDevice 以解析目录后面的内容。若发现路径中的某个节点所在的文件卷尚未安装,则会调用安装文件卷的过程。

4. 文件的创建

文件的创建包含两层意思:创建与打开。因此所谓文件的创建不仅仅是创建文件,还要打开文件。FAT32 文件系统对于文件的创建有两种方式:

- **给定全路径名**:例如"\??\C:\Windows\System32\Notepad.exe"。注意,"\??\"是 NT 格式路径名的特定标识,代表了文件系统的根目录,内核只能识别 NT 格式的路径名。
- **给定一个已打开的目录和相对路径名**:例如已打开目录"\??\C:\Windows",目录后面的相对路径名是"System32\Notepad.exe"。

下面我们分别来介绍这两种方式的文件创建过程。先来看第一种方式,其文件创建步骤如下:

(1) Windows API 调用内核的 NtCreateFile 方法创建文件,内核函数调用对象管理器中的 ObjLookupObjectName 处理和解析文件路径。

(2) 解析文件路径名,当解析到路径"\C:"时,由于上述路径是符号链接,因此对象管理器调用 ObjParseSymbolicLink 方法进行解析,并且认为这是一个文件卷设备符号,故将这个符号链接解析成目标设备对象"Device\Harddisk0\Partition0",并调用设备对象解析函数 IoParseFile 继续解析后面的内容。注意,"Device\Harddisk0\Partition0"这个设备对象的 VPB 指针的 DeviceObject 域已指向文件卷设备对象,代表这个设备已经安装了文件系统。

(3) 调用设备对象解析函数 IopParseFile 对"Windows\System32\Notepad.exe"(后续的路径名)进行解析。

(4) 解析完成后对象管理器将文件创建请求连同文件卷设备对象以 IRP(IRP_MJ_CREATE)的形式下发给 FAT32 文件系统驱动。

(5) FAT32 文件系统调用文件创建的功能函数 VfatCreate 进行文件创建。

第二种方式的创建步骤就要稍微复杂一点,如下所示:

（1）针对已经打开的目录（“\??\C:\Windows”），获取其 FILE_OBJECT，这是个文件对象，代表了对该目录的一次打开操作的上下文。这个数据结构的 VPB 指针指向 VPB，VPB 的 DeviceObject 域指向打开的具体文件卷的设备对象。

（2）对象管理器调用 IopParseDevice 对剩余的相对路径名（“System32\Notepad. exe”）进行解析。解析时以步骤（1）中的 DeviceObject（即文件卷对象）为参数。

（3）对象管理器将文件创建请求连同文件卷设备对象以 IRP 的形式下发给文件系统驱动，这就与第一种方式一致了。

（4）FAT32 文件系统调用文件创建的功能函数 VfatCreate 进行文件创建。

上述两种方式中虽然给定的路径名不同，但是基本思想都是一致的，即都是以文件卷设备对象为基础调用 FAT32 文件系统的文件创建功能函数。我们解析文件路径时不能直接跑到文件卷中进行解析，而应借助文件卷对应的文件系统驱动实现解析，这个实现解析的功能函数就是 VfatCreate。

VfatCreate 调用 fastfat. sys 中的 VfatCreateFile，其中 VfatCreateFile 的参数 DeviceObject 就是从 IRP 传下来的代表具体文件卷的设备对象，而非文件系统驱动的控制设备对象（CDO）。其处理流程是这样的：

（1）针对“\??\C:\Windows\System32”这一个路径，调用 fastfat. sys 中的 VfatOpenFile 方法进行处理，以尝试打开这个文件路径。

（2）如果上述打开操作失败，则证明目标目录项尚不存在，这时需要在目标文件所在的目录中增加一个目录项，并创建文件控制块（FCB），这个 FCB 代表了对该目录的打开，然后再次调用 VfatOpenFile 方法。

5. 文件的读写

下面以读文件为例来讲述 FAT32 文件系统读写的具体步骤，这也是系统调用 NtReadFile 的执行过程：

（1）I/O 管理器通过文件句柄找到 FILE_OBJECT，进而获得 FCB 和 VPB。从 VPB 的 DeviceObject 域中获取读目标文件所属文件卷的设备对象。之所以能先找到 FILE_OBJECT，是因为读写文件时文件必定已经打开了，FILE_OBJECT 是存在的。

（2）I/O 管理器下发主功能码为 IRP_MJ_READ 的 IRP 到文件系统驱动 fastfat. sys，文件系统对应的主功能函数是 VfatDispatchRequest，它调用 VfatRead。VfatRead 的参数包括目标文件的 FCB、文件卷对象、读取数据的偏移和长度、每个扇区的字节数等。

（3）VfatRead 判断是否从缓存读出：

➢ 若缓存中有对应的数据，则调用缓存管理器的读取函数读出；

➢ 否则，从磁盘文件读出（VfatReadFileData），方法是向磁盘驱动下发读取指令。

在处理文件读写请求时，FAT32 文件系统一般会使用缓存管理器来提高读写性能和简化 I/O 操作。

18.3　NTFS 文件系统

NTFS(New Technology File System,新技术文件系统)是 Windows 独有的文件系统,采用了“一切皆文件”的设计思想,其基本原则与特点包括:

➢ 以主文件表(Master File Table,MFT)为核心。
➢ 磁盘上的任何对象包括目录都是文件,每个文件都有一个或多个文件记录来管理这些文件。
➢ 与文件相关的项都被认为是属性,属性分为常驻属性和非常驻属性两种。
➢ 簇是 NTFS 管理的基本单元,就算文件只有一个字节也占用一个簇。
➢ 流是 NTFS 存储的基本单元。
➢ 具有良好的安全性,可以屏蔽未授权用户的访问。
➢ 具有良好的可靠性,可以从意外失败中恢复数据,采用事务的方式保证操作原子性。
➢ 支持长文件名,支持长达 255 B 的文件名,跳出了 8.3 命名规则的束缚。
➢ 具有严格的一致性检查,NTFS 卷上的每个文件都有一个称为文件引用号(也称文件索引号)的 64 位的唯一标识。文件引用号由两部分组成:文件号和文件顺序号。文件号占 48 位,对应于该文件在 MFT 中的位置;文件顺序号随着每次文件记录的重用而增加。

表 18－1 列出了 NTFS、FAT32 与 exFAT 三种文件系统特性的对比。

表 18－1　NTFS、FAT32 与 exFAT 文件系统特性的对比

文件系统	支持的最大分区	支持的最大文件	簇大小	操作系统支持度
NTFS	2 TB	无限制	512 B/1 KB/2 KB/4 KB/8 KB/16 KB/32 KB/64 KB	Windows XP 及以上版本
FAT32	2 TB	4 GB	4 KB/8 KB/16 KB/32 KB/64 KB	Windows XP 及以上版本
exFAT	16 EB	16 EB	512 B～32 MB	Windows Vista/Windows 8

NTFS 卷的结构如图 18－16 所示,主要包括 NTFS 引导扇区、主文件表(MFT)、文件数据和 MFT 的副本。与 FAT32 卷相比,NTFS 卷的首部也是引导扇区,亦位于卷的第一个逻辑扇区,包括了当前卷的布局结构、文件系统的结构、引导代码等信息。

NTFS引导扇区	主文件表(MFT)	文件数据	主文件表(MFT)的副本

图 18－16　NTFS 卷结构

NTFS 文件系统的引导扇区包含了以下内容,如图 18－17 所示:

➢ 跳转指令,用于跳转到启动代码,共 3 B;



➢ NTFS 标志字符串（NTFS）+4 个空字符，共 8 B；
➢ 基础输入输出系统参数块（BPB）和扩展 BPB，共 73 B，这部分数据描述了卷和文件系统的详细信息，例如扇区字节数、簇扇区数、存储介质类型、主文件表及其副本的逻辑簇编号等；
➢ 启动代码，426 B；
➢ 结束标志符，2 B。

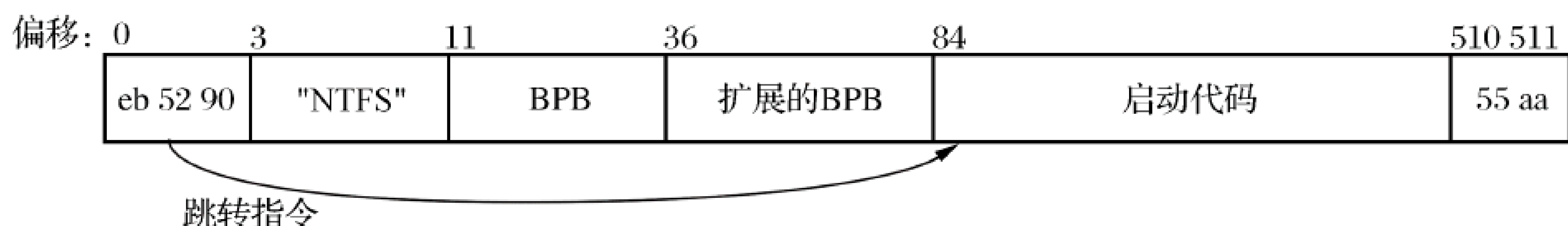


图 18－17　NTFS 文件系统的 DBR 结构

由此可见，NTFS 文件系统的引导扇区与 FAT32 文件系统的引导扇区在布局上基本一致。注意，图 18－16 中的 NTFS 卷结构只是示意图，实际上 MFT 可以在卷上的任意位置，一般由引导扇区的 BPB 提供 MFT 的真实地址。

1．主文件表

NTFS 文件系统的核心是主文件表（MFT）。MFT 本质上是一个关系型数据库，在卷上的任何一个文件，包括 MFT 自己，都在 MFT 中有一条记录，其中 MFT 的第一条记录就是描述它自己的。除了 MFT 本身，在文件系统格式化时，还会创建其他一些元数据，包括日志信息、主文件表副本、卷信息、安全文件等，包括 MFT 自身在内的这些元数据占据 MFT 的前 16 条记录，从第 24 条记录开始才是用户文件/目录的元信息记录。主文件表的前 16 条记录由“＄”符号开头，因此是隐藏的，是不可被用户态进程访问的，如表 18－2 所示。

表 18－2　MFT 中的元数据、用户文件数据及其索引

序号	数据标识	定义
0	$MFT	主文件表本身
1	$MFTMIRR	主文件表的部分镜像
2	$LOGFILE	日志文件
3	$VOLUME	卷文件
4	$ATTRDEF	属性定义列表
5	$ROOT	根目录
6	$BITMAP	位图文件
7	$BOOT	引导文件
8	$BADCLUS	坏簇文件
9	$SECURE	安全文件
10	$UPCASE	大写文件

续表 18－2

序号	数据标识	定义
11	$EXTEND METADATA DIRECTORY	扩展元数据目录
12	$EXTEND \ $REPARSE	重解析点文件
13	$EXTEND \ $USNJRNL	变更日志文件
14	$EXTEND \ $QUOTA	配额管理文件
15	$EXTEND \ $OBJID	对象 ID 文件
16～23		保留
23＋		用户文件和目录

其中元数据 $ROOT 保存了存放于该卷的根目录下的所有文件和目录的索引。在访问了一个文件以后，NTFS 就保存该文件的 MFT 的引用，以便于第二次直接访问使用。而元数据 $BITMAP 中的每一位代表了卷中的一个簇，可以被扩大，因此 NTFS 卷可以被很方便地扩展。

MFT 中的元数据是非常重要的，因此应该对其进行备份存储，其位置大约在数据区域的中间部位，如图 18－18 所示。

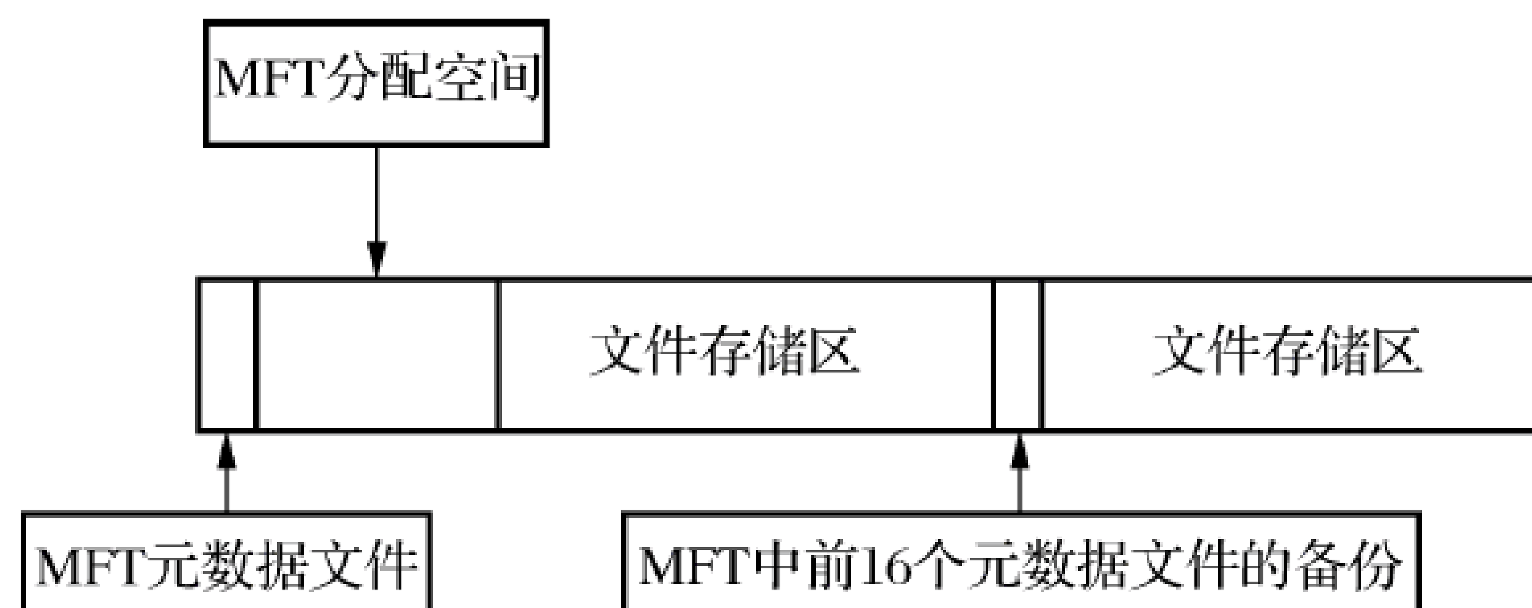


图 18－18　MFT 中重要元数据的备份

NTFS 把磁盘分成了两大部分，其中大约 12% 被分配给了 MFT，以满足其不断增长的文件数量；余下的 88% 的空间被分配用来存储文件。当文件耗尽了存储空间时，操作系统会简单地减少 MFT 空间，并把省下来的这部分空间分配给文件存储。当有剩余空间时，这些空间又会重新划分给 MFT。虽然系统尽力保持 MFT 空间的专用性，但是有时不得不做出牺牲。

每个文件或目录在 NTFS 中都是一组“属性–值”对，包括时间戳、访问模式、链接到该文件的数量等通用信息，也包括文件名、属性列表、对象标识符、安全描述符、扩展属性等非通用性信息，更包括数据属性。

在 NTFS 中，一个文件的内容都被定义为数据属性。从某种意义上来看，NTFS 中的文件就像 JSON 格式的数据一样，每一项都由属性头和属性体构成。不过也不是所有的属性都在 MFT 中，因为一条 MFT 记录的大小是 1 KB，如果文件的数据属性和其他属性加起来不超过 1 KB 自然可以放在一条记录中，但如果是个大文件，其总大小超过 1 KB，自然会溢出到记录

之外。我们将 MFT 记录之内的属性称为“驻留属性”,溢出的属性称为“非驻留属性”,前者一般包括文件名、时间戳等;后者主要是数据属性。图 18－19 展示了 MFT 的记录格式与驻留属性、非驻留属性,表 18－3 则列出了 NTFS 预定义的属性类型。

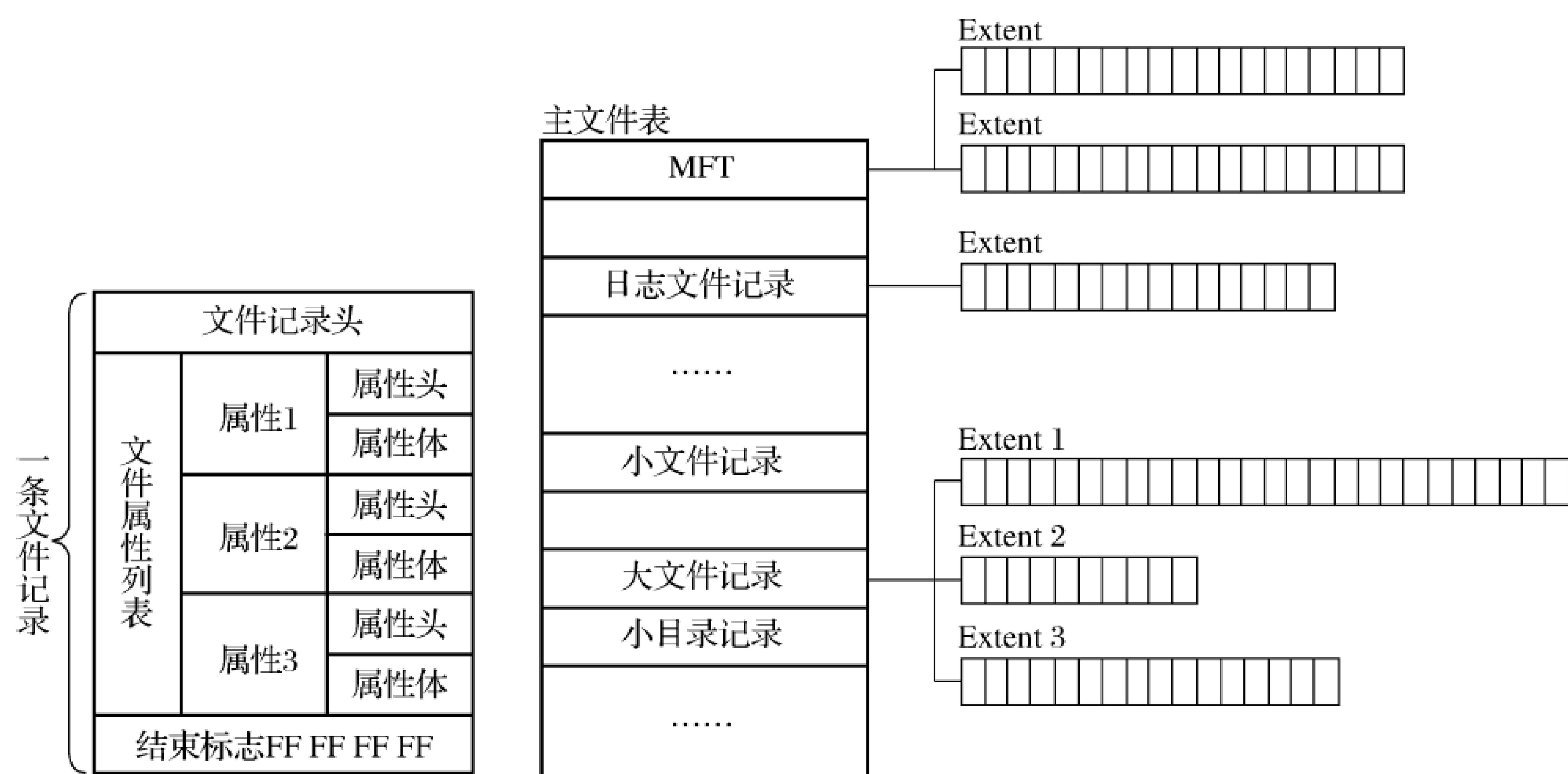


图 18－19　MFT 文件记录的格式与驻留属性、非驻留属性

表 18－3　NTFS 预定义的属性类型

属性名	属性描述
$VOLUME_INFORMATION	卷信息属性,仅存在于 $VOLUME 元数据文件中
$VOLUME_NAME	卷名称,仅存在于 $VOLUME 元数据文件中
$STANDARD_INFORMATION	标准属性,这包括基本文件属性,如只读、存档;时间标记,如文件的创建时间和最近一次修改的时间;有多少目录指向本文件,也就是它的硬链接数(Hard Link Count)
$FILE_NAME	文件名属性,使用 U Unicode 字符表示。由于 MS-DOS 不能正确识别 Win32 子系统创建的文件名,当 Win32 子系统创建一个文件名时,NTFS 会自动生成一个备用的 MS-DOS 文件名,所以一个文件有多种文件名属性
$SECURITY_DESCRIPTOR	安全描述符属性,描述了文件系统的访问控制安全属性。主要用于保护文件以防止未授权访问
$DATA	文件的数据属性,也就是文件的内容(在 NTFS 文件系统中,一个文件除了支持文件数据即未命名的属性外,还可支持其他命名属性,即可以有多个数据属性,目录没有默认的数据属性,但是有可选的命名数据属性)
$INDEX_ROOT	索引根属性
$INDEX_ALLOCATION	索引分配属性

续表 18－3

属性名	属性描述
$BITMAP	位图属性
$ATTRIBUTE_LIST	属性列表，当一个文件需要使用多条 MFT 记录时用来表示该文件的属性列表
$OBJECT_ID	对象 ID，64 字节的标识符，低 16 字节用于描述文件或目标标识
$REFARSE_POINT	重点解析属性
$EA	扩充属性，主要为了与 OS/2 兼容，现已使用不多
$EA_INFORMATION	扩充属性信息，主要为了与 OS/2 兼容，现已使用不多
$LOGGED_UTILITY_STREAM	EFS 加密属性，主要为实现 EFS(Encryped File System，加密文件系统)而存储相关加密信息，如解码密钥、合法访问的用户列表等

不是每一条记录都有相同的属性，记录之间只是有一些共性的属性是必须要有的，每一条记录可以根据自己的数据特性选择属性类别。当包括数据属性在内的所有属性都在一条记录中时，从 MFT 就可以读取文件数据了；但如果还有非驻留属性，则要进一步访问其他扇区，非驻留属性一般是数据属性。非驻留属性被组织为一些簇，但这些簇就不在 MFT 中了，而是在 NTFS 卷的数据区域中(文件数据部分)。

2. NTFS 数据区域

NTFS 数据区域中的簇按照 RUN(串)的概念来组织。所谓 RUN，就是指属性中一段连续的字节范围在磁盘上占据的逻辑簇，用 <VCN，LCN，LengthOfRun> 这种记录结构来表示。NTFS 使用逻辑簇号(Logical Cluster Number，LCN)和虚拟簇号(Virtual Cluster Number，VCN)来对簇进行定位。

- **VCN**：是对属于特定文件的簇从头到尾所进行的顺序编号，以便于引用文件中的数据。VCN 表示某文件/目录在所占空间中的簇编号，每个文件的 VCN 都是从 0 开始编号的。VCN 可以映射成 LCN，并且不必要求在物理空间上连续。
- **LCN**：是对整个卷中所有的簇从头到尾所进行的顺序编号，表示某文件/目录所占空间的簇在卷中的簇编号，不一定是从 0 开始的。
- **LengthOfRun**：本串的长度，即簇的个数。

当某个属性是非驻留的，则在 MFT 的文件记录中该属性的属性头会标识出一个或多个行串，如图 18－20 所示。访问文件时首先判断属性头的行串信息，判断该属性是否驻留，从而采用不同的方式访问整个数据。注意，每一个行串中的 LCN 是连续的。

还有一种情况，即文件中的属性太多，一条记录放不下了。这种情况也会导致溢出，此时可以使用 $ATTRIBUTE_LIST 结构来存放这些属性。$ATTRIBUTE_LIST 包含了该文件各个属性的名称和类型代码，以及该属性所在 MFT 记录的编号。采用多条文件记录来存放这些属性，其中第一条 MFT 记录被称为基本文件记录。

NTFS 中每个属性由单个的流(stream)组成，称为属性流。流是简单的字符队列，NTFS

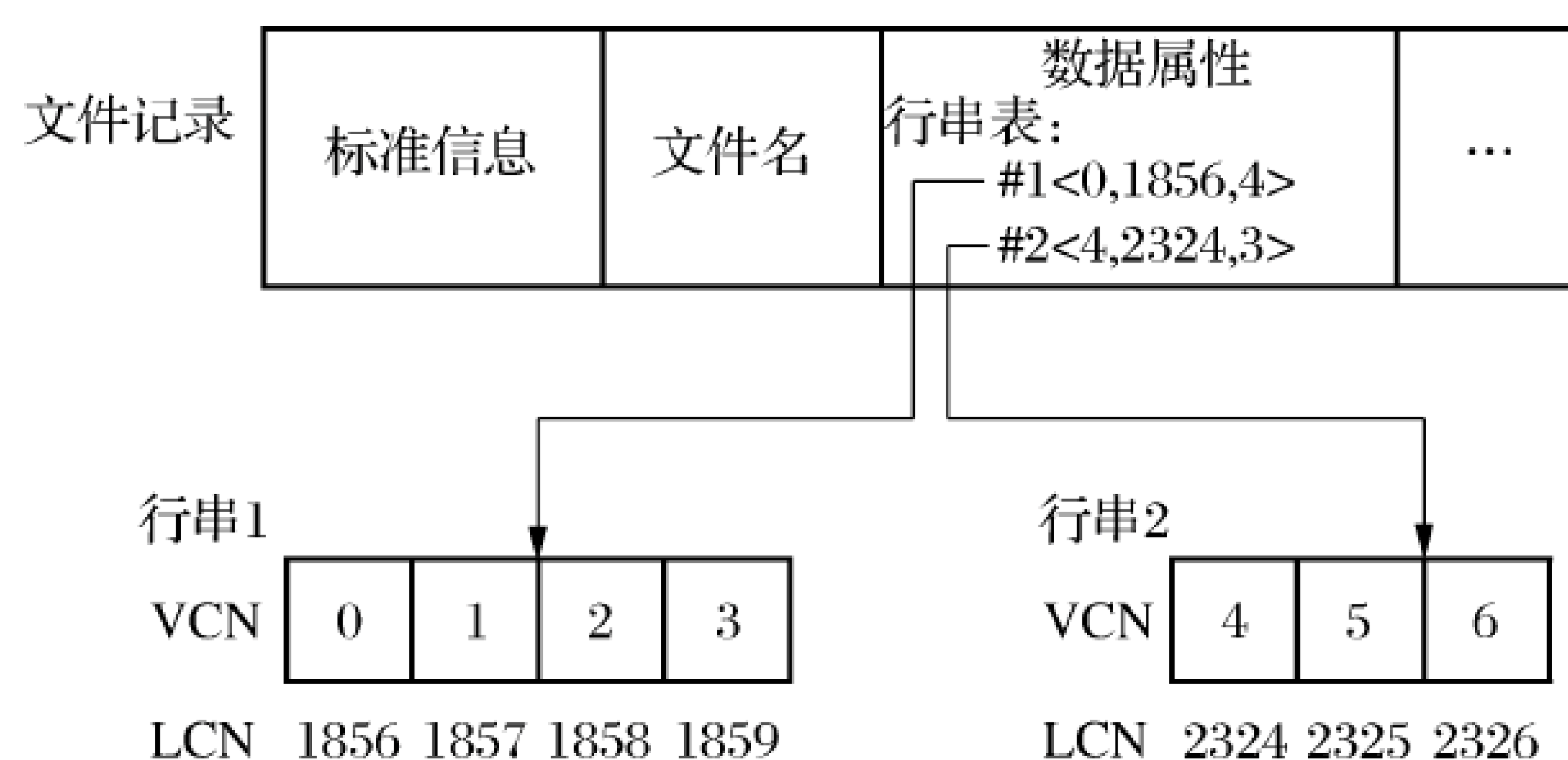


图 18-20　非驻留属性的行串信息

提供对属性流的各种操作:创建、删除、读取、写入等。读写操作一般是针对文件的未命名属性的,对于已命名属性则可以通过已命名的属性流句法来进行操作。NTFS 属性流本是 NTFS 文件格式中的一种正常特性,但却可以被一些木马病毒所利用,而杀毒软件对 NTFS 属性流文件的检测能力十分薄弱,很多木马病毒就趁此机会利用 NTFS 属性流将自己完全隐藏在系统中。

NTFS 的目录只是一个简单的文件名和文件引用号的索引,如果数据量不大自然可以在一个记录中存储;如果目录中文件和子目录的数量很大,则需要以 B + 树的方式存储这些文件和子目录信息以便于快速查找。图 18-21 和 18-22 是两个目录记录结构的示例。



图 18-21　小目录的目录记录结构

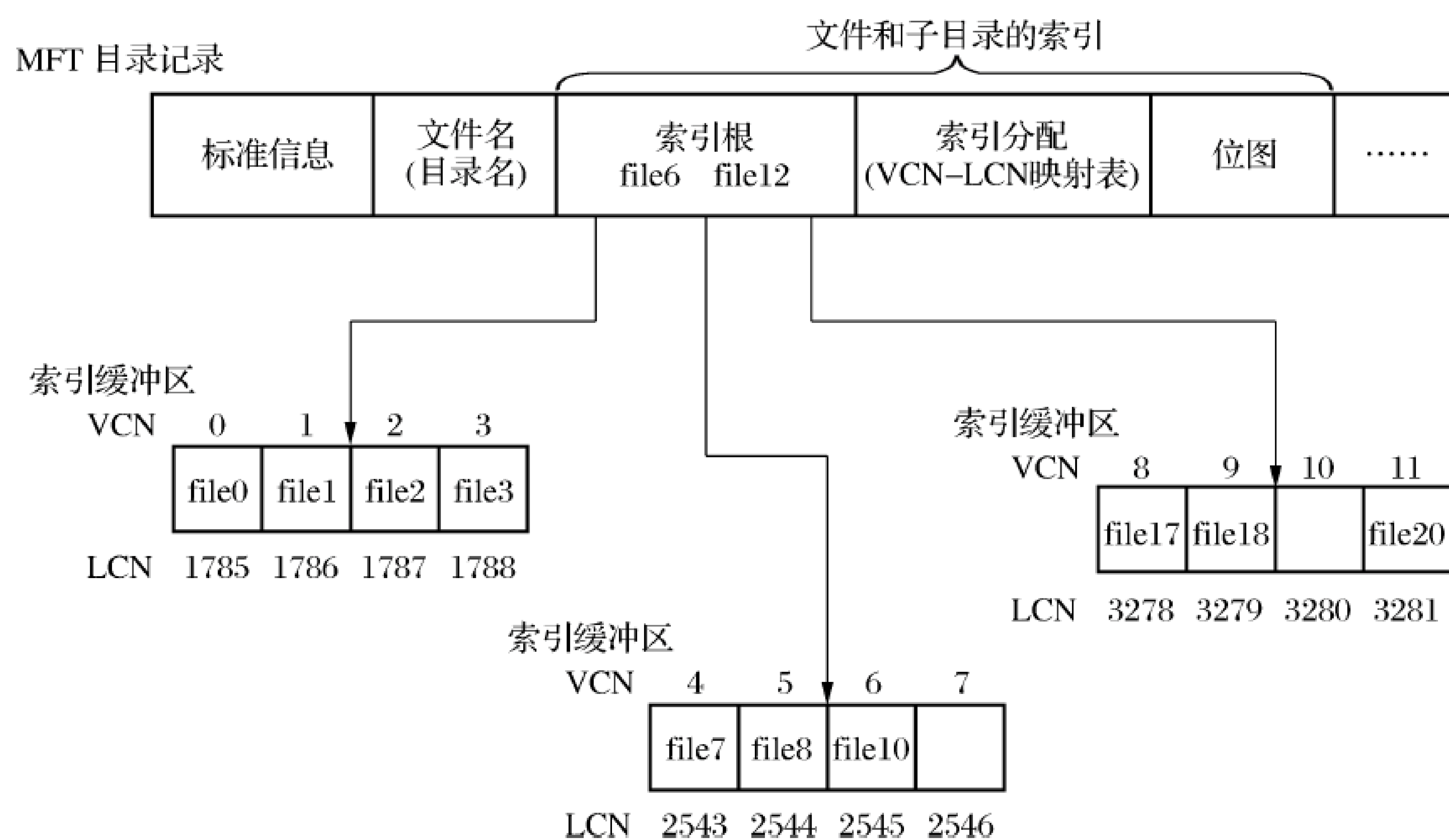


图 18-22　大目录的目录记录结构

3. NTFS 驱动模块

NTFS 文件系统的驱动承载映像文件是 ntfs. sys。与 FAT32 文件系统的 fastfat. sys 模块类似，ntfs. sys 也是一个 NT 式驱动模块，其依赖的模块如图 18 – 23 所示。

ntfs.sys

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ntoskrnl.exe	439	001841D8	00000000	00000000	001841C4	00045000
msrpc.sys	5	00184F98	00000000	00000000	001841B8	00045DC0
CLFS.SYS	37	00184FC8	00000000	00000000	001841AC	00045DF0
ksecdd.sys	10	001850F8	00000000	00000000	001841A0	00045F20

图 18 – 23　ntfs. sys 的依赖模块

在 NTFS 文件系统中，DriverEntry 依然是驱动的入口函数，但 ntfs. sys 没有 AddDevice 函数。在 DriverEntry 中主要完成以下工作：

➢ 创建控制设备对象 Ntfs，对象类型为 FILE_DEVICE_DISK_FILE_SYSTEM，这与 FAT32 创建的设备对象的类型是一致的，并将设备对象的扩展部分定义为 NTFS_GLOBAL_DATA 数据结构。

➢ 初始化驱动对象的主功能函数，具体如下：

- IRP_MJ_CREATE：NtfsCreate；
- IRP_MJ_CLOSE：NtfsClose；
- IRP_MJ_READ：NtfsRead；
- IRP_MJ_WRITE：NtfsWrite；
- IRP_MJ_FILE_SYSTEM_CONTROL：NtfsFileSystemControl；
- IRP_MJ_DIRECTORY_CONTROL：NtfsDirectoryControl；
- IRP_MJ_QUERY_INFORMATION：NtfsQueryInformation；
- IRP_MJ_QUERY_VOLUME_INFORMATION：NtfsQueryVolumeInformation；
- IRP_MJ_SET_VOLUME_INFORMATION：NtfsSetVolumeInformation。

➢ 通过 IoRegisterFileSystem 方法向全局列表注册 NTFS 文件系统。

当没有任何卷采用 NTFS 文件系统时，Windows 并没有加载 NTFS 文件系统驱动，此时称 NTFS 未激活。当有一个新的使用了 NTFS 文件的卷被加载时，NTFS 文件系统驱动也就被加载了，此时称 NTFS 被激活。

图 18 – 24 是 NTFS 驱动的功能函数和快速处理函数，可以看出，大部分函数都位于 ntfs. sys 模块中，而诸如 IRP_MJ_CREATE_NAMED_PIPE、IRP_MJ_CREATE_MAILSLOT 等功能函数则位于内核模块 ntoskrnl. sys 中，这说明包括命名管道、邮件槽等设备的创建已经从文件系统中剥离了出来而被归为了内核的一部分。

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
28	(Ntfs)IRP_MJ_CREATE	0xFFFFF880012C1370	-	0xFFFFF880012C1370	C:\Windows\System32\Drivers\Ntfs.sys
29	(Ntfs)IRP_MJ_CREATE_NAMED_PIPE	0xFFFFF800046BBED0	-	0xFFFFF800046BBED0	C:\Windows\system32\ntoskrnl.exe
30	(Ntfs)IRP_MJ_CLOSE	0xFFFFF880012BD6B0	-	0xFFFFF880012BD6B0	C:\Windows\System32\Drivers\Ntfs.sys
31	(Ntfs)IRP_MJ_READ	0xFFFFF880012272A0	-	0xFFFFF880012272A0	C:\Windows\System32\Drivers\Ntfs.sys
32	(Ntfs)IRP_MJ_WRITE	0xFFFFF88001232930	-	0xFFFFF88001232930	C:\Windows\System32\Drivers\Ntfs.sys
33	(Ntfs)IRP_MJ_QUERY_INFORMATION	0xFFFFF880012A6FE0	-	0xFFFFF880012A6FE0	C:\Windows\System32\Drivers\Ntfs.sys
34	(Ntfs)IRP_MJ_SET_INFORMATION	0xFFFFF88001227E90	-	0xFFFFF88001227E90	C:\Windows\System32\Drivers\Ntfs.sys
35	(Ntfs)IRP_MJ_QUERY_EA	0xFFFFF880012A6FE0	-	0xFFFFF880012A6FE0	C:\Windows\System32\Drivers\Ntfs.sys
36	(Ntfs)IRP_MJ_SET_EA	0xFFFFF880012A6FE0	-	0xFFFFF880012A6FE0	C:\Windows\System32\Drivers\Ntfs.sys
37	(Ntfs)IRP_MJ_FLUSH_BUFFERS	0xFFFFF8800129E3F0	-	0xFFFFF8800129E3F0	C:\Windows\System32\Drivers\Ntfs.sys
38	(Ntfs)IRP_MJ_QUERY_VOLUME_INFORMATION	0xFFFFF880012A7380	-	0xFFFFF880012A7380	C:\Windows\System32\Drivers\Ntfs.sys
39	(Ntfs)IRP_MJ_SET_VOLUME_INFORMATION	0xFFFFF880012A7380	-	0xFFFFF880012A7380	C:\Windows\System32\Drivers\Ntfs.sys
40	(Ntfs)IRP_MJ_DIRECTORY_CONTROL	0xFFFFF880012B2A40	-	0xFFFFF880012B2A40	C:\Windows\System32\Drivers\Ntfs.sys
41	(Ntfs)IRP_MJ_FILE_SYSTEM_CONTROL	0xFFFFF880012D3A80	-	0xFFFFF880012D3A80	C:\Windows\System32\Drivers\Ntfs.sys
42	(Ntfs)IRP_MJ_DEVICE_CONTROL	0xFFFFF8800129CDF0	-	0xFFFFF8800129CDF0	C:\Windows\System32\Drivers\Ntfs.sys
43	(Ntfs)IRP_MJ_INTERNAL_DEVICE_CONTROL	0xFFFFF800046BBED0	-	0xFFFFF800046BBED0	C:\Windows\system32\ntoskrnl.exe
44	(Ntfs)IRP_MJ_SHUTDOWN	0xFFFFF8800139B870	-	0xFFFFF8800139B870	C:\Windows\System32\Drivers\Ntfs.sys
45	(Ntfs)IRP_MJ_LOCK_CONTROL	0xFFFFF8800125AAF0	-	0xFFFFF8800125AAF0	C:\Windows\System32\Drivers\Ntfs.sys
46	(Ntfs)IRP_MJ_CLEANUP	0xFFFFF880012C6440	-	0xFFFFF880012C6440	C:\Windows\System32\Drivers\Ntfs.sys
47	(Ntfs)IRP_MJ_CREATE_MAILSLOT	0xFFFFF800046BBED0	-	0xFFFFF800046BBED0	C:\Windows\system32\ntoskrnl.exe
48	(Ntfs)IRP_MJ_QUERY_SECURITY	0xFFFFF880012A7380	-	0xFFFFF880012A7380	C:\Windows\System32\Drivers\Ntfs.sys
49	(Ntfs)IRP_MJ_SET_SECURITY	0xFFFFF880012A7380	-	0xFFFFF880012A7380	C:\Windows\System32\Drivers\Ntfs.sys
50	(Ntfs)IRP_MJ_POWER	0xFFFFF800046BBED0	-	0xFFFFF800046BBED0	C:\Windows\system32\ntoskrnl.exe
51	(Ntfs)IRP_MJ_SYSTEM_CONTROL	0xFFFFF800046BBED0	-	0xFFFFF800046BBED0	C:\Windows\system32\ntoskrnl.exe
52	(Ntfs)IRP_MJ_DEVICE_CHANGE	0xFFFFF800046BBED0	-	0xFFFFF800046BBED0	C:\Windows\system32\ntoskrnl.exe
53	(Ntfs)IRP_MJ_QUERY_QUOTA	0xFFFFF880012A6FE0	-	0xFFFFF880012A6FE0	C:\Windows\System32\Drivers\Ntfs.sys
54	(Ntfs)IRP_MJ_SET_QUOTA	0xFFFFF880012A6FE0	-	0xFFFFF880012A6FE0	C:\Windows\System32\Drivers\Ntfs.sys
55	(Ntfs)IRP_MJ_PNP_POWER	0xFFFFF880012EC2B0	-	0xFFFFF880012EC2B0	C:\Windows\System32\Drivers\Ntfs.sys
27[FastIo]	(Ntfs)FastIoCheckIfPossible	0xFFFFF88001349D30	-	0xFFFFF88001349D30	C:\Windows\System32\Drivers\Ntfs.sys
28[FastIo]	(Ntfs)FastIoRead	0xFFFFF880012B2FC0	-	0xFFFFF880012B2FC0	C:\Windows\System32\Drivers\Ntfs.sys
29[FastIo]	(Ntfs)FastIoWrite	0xFFFFF880012B5CE0	-	0xFFFFF880012B5CE0	C:\Windows\System32\Drivers\Ntfs.sys
30[FastIo]	(Ntfs)FastIoQueryBasicInfo	0xFFFFF880012A77A0	-	0xFFFFF880012A77A0	C:\Windows\System32\Drivers\Ntfs.sys
31[FastIo]	(Ntfs)FastIoQueryStandardInfo	0xFFFFF880012A7010	-	0xFFFFF880012A7010	C:\Windows\System32\Drivers\Ntfs.sys
32[FastIo]	(Ntfs)FastIoLock	0xFFFFF880012960D0	-	0xFFFFF880012960D0	C:\Windows\System32\Drivers\Ntfs.sys
33[FastIo]	(Ntfs)FastIoUnlockSingle	0xFFFFF88001295EA0	-	0xFFFFF88001295EA0	C:\Windows\System32\Drivers\Ntfs.sys
34[FastIo]	(Ntfs)FastIoUnlockAll	0xFFFFF8800134A050	-	0xFFFFF8800134A050	C:\Windows\System32\Drivers\Ntfs.sys
35[FastIo]	(Ntfs)FastIoUnlockAllByKey	0xFFFFF88001349DF0	-	0xFFFFF88001349DF0	C:\Windows\System32\Drivers\Ntfs.sys
38[FastIo]	(Ntfs)ReleaseFileForNtCreateSection	0xFFFFF88001227E10	-	0xFFFFF88001227E10	C:\Windows\System32\Drivers\Ntfs.sys
40[FastIo]	(Ntfs)FastIoQueryNetworkOpenInfo	0xFFFFF8800129B440	-	0xFFFFF8800129B440	C:\Windows\System32\Drivers\Ntfs.sys
41[FastIo]	(Ntfs)AcquireForModWrite	0xFFFFF88001235910	-	0xFFFFF88001235910	C:\Windows\System32\Drivers\Ntfs.sys
42[FastIo]	(Ntfs)MdlRead	0xFFFFF880012972D0	-	0xFFFFF880012972D0	C:\Windows\System32\Drivers\Ntfs.sys
43[FastIo]	(Ntfs)MdlReadComplete	0xFFFFF800046B0DAC	-	0xFFFFF800046B0DAC	C:\Windows\system32\ntoskrnl.exe
44[FastIo]	(Ntfs)PrepareMdlWrite	0xFFFFF880012974B0	-	0xFFFFF880012974B0	C:\Windows\System32\Drivers\Ntfs.sys
45[FastIo]	(Ntfs)MdlWriteComplete	0xFFFFF80004962128	-	0xFFFFF80004962128	C:\Windows\system32\ntoskrnl.exe
50[FastIo]	(Ntfs)FastIoQueryOpen	0xFFFFF880012A1250	-	0xFFFFF880012A1250	C:\Windows\System32\Drivers\Ntfs.sys
51[FastIo]	(Ntfs)ReleaseForModWrite	0xFFFFF880012359B0	-	0xFFFFF880012359B0	C:\Windows\System32\Drivers\Ntfs.sys
52[FastIo]	(Ntfs)AcquireForCcFlush	0xFFFFF880012375A0	-	0xFFFFF880012375A0	C:\Windows\System32\Drivers\Ntfs.sys
53[FastIo]	(Ntfs)ReleaseForCcFlush	0xFFFFF880012375F0	-	0xFFFFF880012375F0	C:\Windows\System32\Drivers\Ntfs.sys

图 18－24　NTFS 驱动各功能函数和快速处理函数

4．文件的读取

下面我们以一个实例来讲解 NTFS 读取文件的过程。假设文件的路径仍然是“C:\Windows\System32\Notepad.exe”。要访问这个 NTFS 文件的数据，首先应在 $MFT 文件中找到与该路径对应的文件记录，然后在找到的文件记录中查找 $Data 属性。若 $Data 属性驻留，则直接读出其数据；若 $Data 属性非驻留，则到 Runlist 指出的位置读出文件的数据。文件索引的目标便是根据文件名找到该文件的文件记录在 $MFT 中的位置。每个目录 B＋树节点的每个 Key 至少包含两条信息：文件名和该文件的文件记录在 $MFT 文件中的序号（第几条文件记录）。具体步骤如下：

（1）首先在 $MFT 文件中读取第 5 条 FileRecord，该 FileRecord 就是 $Boot 文件的文件记录。$Boot 文件中有 $MFT 文件的位置，Windows 启动时已经被读取到内存中了。

（2）系统解析该文件记录时发现该记录为目录文件的记录，便读取 $INDEX_ALLOCATION 属性的 Runlist 所指区域中的数据，这些数据构成一棵 B＋树。

（3）遍历这棵 B＋树，查找包含文件名“Windows”的 Key，找到后从该 Key 中读取“Windows”文件的文件记录在 $MFT 文件中的序号 n。

（4）回到 $MFT 文件中读取第 n 条 FileRecord，解析后发现“Windows”为目录，便读取 $INDEX_ALLOCATION 属性的 Runlist 所指区域中的数据，这些数据构成一棵 B + 树。

（5）遍历这棵 B + 树，查找包含文件名“System32”的 Key，找到后从该 Key 中读取“System32”文件的文件记录在 $MFT 文件中的序号 m。

（6）回到 $MFT 文件中读取第 m 条 FileRecord，解析后发现“System32”依然为目录，便读取 $INDEX_ALLOCATION 属性的 Runlist 所指区域中的数据，这些数据又构成一棵 B + 树。

（7）遍历这棵 B + 树，查找包含文件名“Notepad. exe”的 Key，找到后从该 Key 中读取“Notepad. exe”文件的文件记录在 $MFT 文件中的序号 i。

（8）回到 $MFT 文件中读取第 i 条 FileRecord，解析后发现“Notepad. exe”为一般文件，便读取 $Data 属性的 runlist 所指向区域中的数据。

经过上述步骤，读取操作便完成了。

18.4　ReFS 文件系统

ReFS（Resilient File System，弹性文件系统）是在 Windows Server 2012/Windows 8 中新引入的一种文件系统，目前只能应用于存储数据，还不能引导系统，并且在移动媒介上也无法使用。

相较于 NTFS 文件系统，ReFS 文件系统提升了更多的可靠性，特别是对于老化的磁盘或者是当机器发生突然断电的时候。ReFS 的可靠性来自底层的变化，比如文件元数据的存储和更新。ReFS 兼容 Storage Spaces 跨区卷技术，当磁盘出现读取和写入失败时，ReFS 会进行系统校验，可以检测到这些错误并进行正确的文件拷贝。

ReFS 类似 NTFS 和 RAID5 的组合，因自带奇偶校验而会损失存储空间，所以 ReFS 会占用硬盘的大部分空间来保证数据完整性，但其可靠性自然要比 NTFS 高很多。

ReFS 文件系统的关键功能如下：

- 提供带有校验和的完整性元数据。
- 提供可选用户数据完整性的完整性流。
- 通过写入时分配事务模型实现可靠的磁盘更新（也称为写时复制）。
- 支持超大规模的卷、文件和目录。
- 存储池和虚拟化特性使得文件系统可建立并易于管理。
- 通过数据条带化技术提高性能（带宽可管理）并通过备份提高容错性。
- 通过磁盘扫描防止潜在的磁盘错误。
- 借助“数据打捞”实现损坏还原，以便在任何情况下尽可能提高卷的可用性。
- 跨计算机共享存储池，以提供额外的容错性和负载平衡。

此外，ReFS 还从 NTFS 继承了某些功能和语义，包括 BitLocker 加密、用于安全的访问控

制列表、USN 日志、更改通知、符号链接、交接点、装入点、重解析点、卷快照、文件 ID 和操作锁等。当然,客户端只要使用任何操作系统中可访问现有 NTFS 卷的文件访问 API 就可以访问以 ReFS 格式存储的数据。

18.5 文件系统识别器

文件系统识别器是个内核态的驱动,其功能就是检查物理存储介质,如果能识别则加载相应的文件系统,正因为如此才会被称为“文件系统识别器”。它的引入使得系统几乎从不需要预先加载所有的文件系统驱动程序,实现了只用一个小驱动就可以完成几百千字节甚至若干兆字节内存才能完成的工作。当文件系统加载完成后,文件系统识别器也就完成了使命,可以卸载了。Windows 中文件系统识别器的承载映像是 fs_rec. sys 模块,其生成的设备对象如图 18－25 所示。

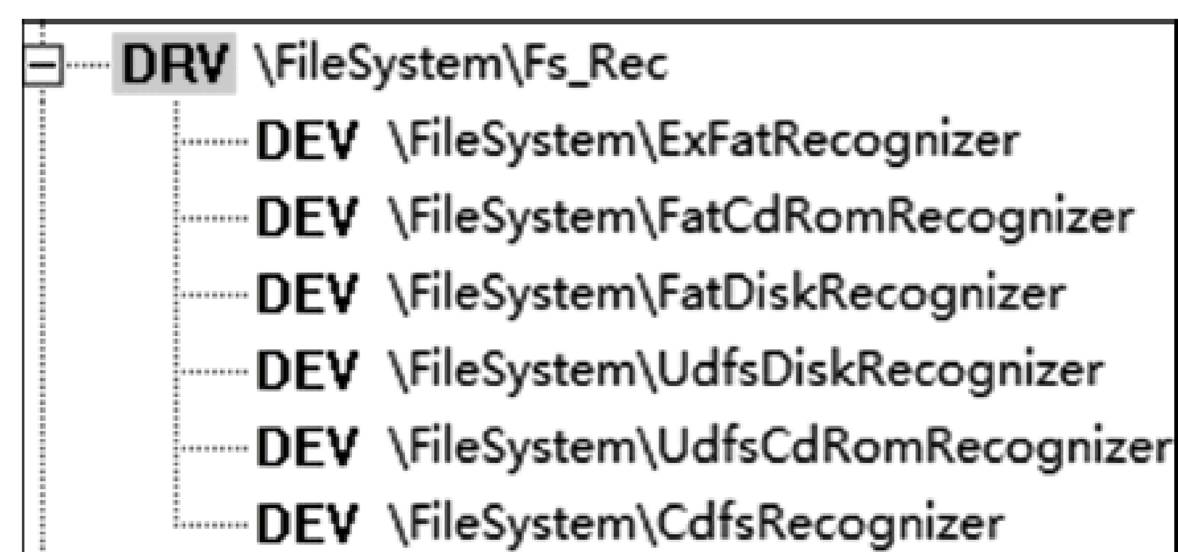


图 18－25　文件系统识别器生成的设备对象

在实际环境中,几乎所有物理存储介质的文件系统都利用了文件系统识别器。其对应的物理存储介质没有被访问,则其文件系统也不会被真正加载。当新的物理存储介质进入系统后,I/O 管理器会依次尝试各种文件系统进行卷识别,如果识别成功,则立刻加载真正的文件系统驱动,而对应的文件系统识别器会被卸载。

文件系统识别器是通过磁盘上的标识符来识别文件系统的。标识符就是一串序列号或者字符串,可能存在于分区表中,也可能存在于其他地方,如表 18－4 所示。

表 18－4　常见的文件系统标识符

文件系统名	文件系统标识符
HFS	0x244
NTFS	" NTFS"
FAT	0xe9 或 0xeb 或 0x49

文件系统识别器实际上就是一个只处理挂载请求的文件系统驱动程序。因此,它基于相应文件系统类型创建设备对象,并向 I/O 管理器注册使用的文件系统,然后等待被调用去挂载卷。如果识别器确认了卷属于它的文件系统则返回状态码 STATUS_FS_DRIVER_REQUIRED,而不是接受这个挂载请求。接下来 I/O 管理器调用识别器,使其加载整个文件系统驱动程序,具体做法是发送主功能码为 IRP_MJ_FILE_SYSTEM_CONTROL、副功能码为 IRP_MN_LOAD_FILE_SYSTEM 的 IRP。

总之,文件系统识别器虽然是一个简单的驱动程序,但它却是必需的。它可以使你的驱动“按需启动”,以最小化内存需求。文件系统识别器就是文件系统驱动的“小替身”。

18.6　文件系统过滤框架 FltMgr

Windows 文件系统框架提供了两类过滤驱动模型:文件系统过滤驱动和文件系统小过滤驱动。前者是较为通用的设备对象堆叠方式的过滤模型;后者则依托于 Windows FltMgr 驱动框架,并且不需要处理设备对象堆叠和挂钩,也不会生成设备对象,使开发者专注于过滤业务本身,因此具有极大的便捷性和稳定性。

1. 文件系统过滤驱动

这种类型的过滤驱动常用于病毒检测、文件加密、文件分析、文件压缩等场景,并且使用的是 Windows 设备驱动框架,入口函数也是 DriverEntry,但没有 AddDevice 方法,也就是说这类过滤驱动是 NT 式驱动,也不处理即插即用和电源管理的命令。这种驱动的过滤目标对象是 IRP。

这类驱动可以利用 I/O 管理器的 IoRegisterFsRegistrationChange 方法来接收文件系统的变化。这个方法将接收方的回调函数指针注册到 I/O 管理器,当注册成功后接收方会接收到系统中现有文件系统的信息回调,每一种文件系统均有一次回调,以使调用方知道目前系统中有多少文件系统。

从图 18 - 26 看出,过滤驱动加入后,在文件系统设备栈的顶端(FltMgr 设备对象的上方)增加了一个新的设备类型,这与普通的基于设备对象堆叠的过滤方式如出一辙。流经 I/O 管理器和 FltMgr 设备对象的 IRP 会经过这个过滤设备对象以便截取过滤。但是这种方式只能将过滤设备堆叠到设备栈的顶端,过滤的位置是受限的。

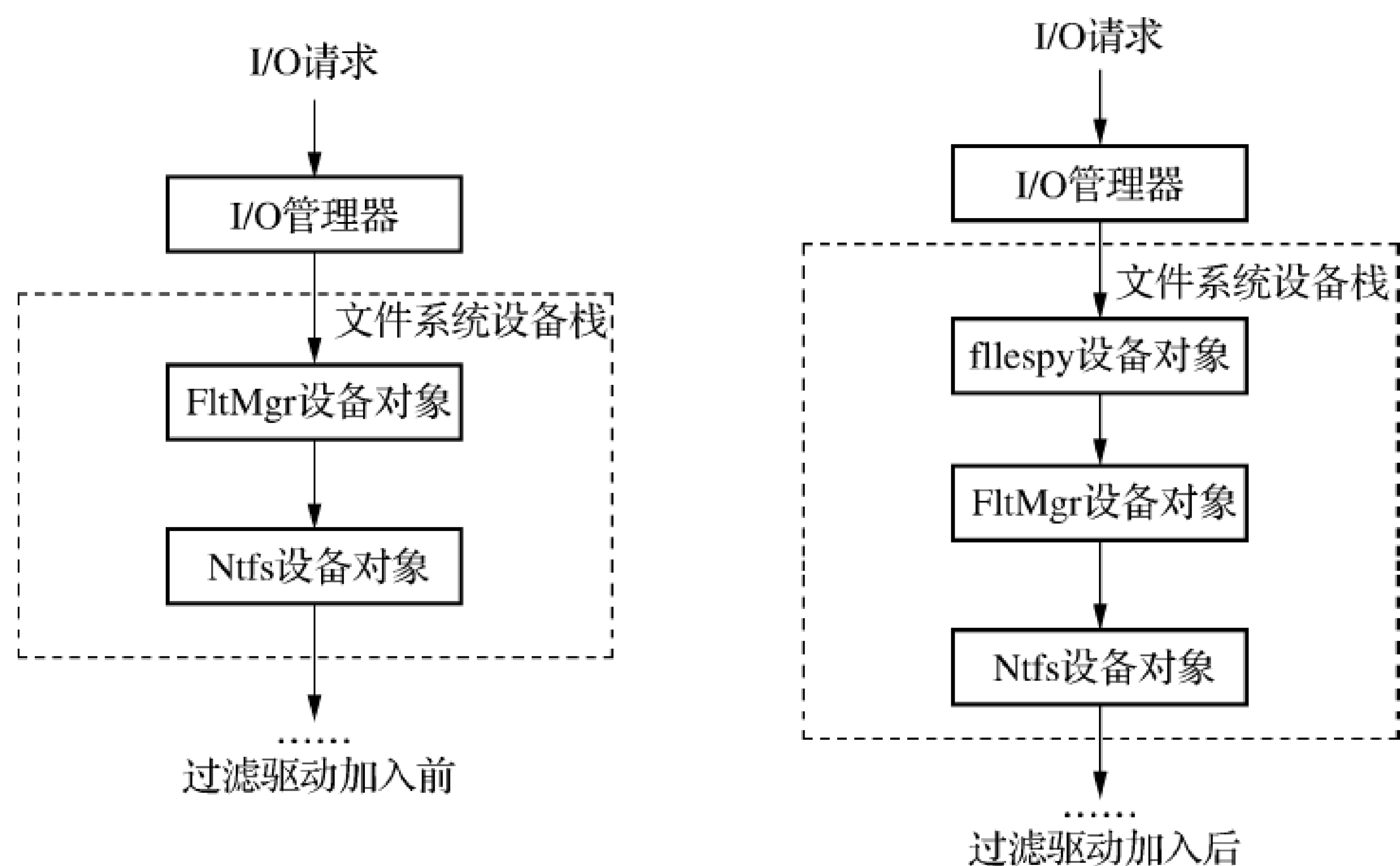


图 18 - 26　过滤驱动加入系统后的设备栈

我们要注意的是,文件系统又不同于普通的设备驱动。因为文件系统一般创建两个设备对象:一个是 CDO,即控制设备对象,这一般是用于挂入系统列表的命名对象;另一个是文件系统卷对象,一般每个卷生成一个卷设备对象,堆叠在卷管理器创建的卷设备之上,且是

未命名的。所有流经文件系统的 IRP 都是针对卷设备对象而言的，虽然在大多数情况下卷设备对象与 CDO 共享驱动的功能函数，但 CDO 承担的任务非常少，在过滤操作中基本可以忽略不计。

堆叠过滤是比较常规的做法，也是比较兴师动众的做法。

2. 文件系统小过滤驱动

文件系统小过滤驱动依附于 FltMgr 驱动框架。FltMgr 是 Windows 文件系统的 I/O 过滤框架，允许文件系统小驱动程序向本框架注册需要过滤拦截的 I/O 操作。这些拦截的目标既包括 IRP，还包括快速 I/O 函数，也包括回调操作。

当系统中没有小过滤驱动时，FltMgr 框架是透明的，并不启用。当有小过滤驱动注册到 FltMgr 框架时框架才会启动。但小过滤驱动并不会生成设备对象堆叠在文件系统设备栈中，这些都由框架完成和实现，小过滤驱动不需要关注。

FltMgr 驱动框架允许小过滤驱动注册“前序操作”和“尾声操作”，分别对应 I/O 操作之前的回调函数和 I/O 操作之后的回调函数。

FltMgr 驱动框架的承载映像文件是 fltMgr. sys，很显然这也是个驱动模块，也有入口函数 DriverEntry，其导出函数如图 18 - 27 所示。可以看出，fltMgr. sys 导出了诸如 FltAttachVolume 这样的卷挂载接口函数以供调用。

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	0003D66C	0003DD3A	0003D9E4	0003E0C4
(nFunctions)	Dword	Word	Dword	szAnsi
00000010	0000AE70	000F	00041A8B	FltAllocatePoolAlignedWithTag
00000011	0000B4A0	0010	00041AA9	FltApplyPriorityInfoThread
00000012	00037020	0011	00041AC4	FltAttachVolume
00000013	00036FF0	0012	00041AD4	FltAttachVolumeAtAltitude
00000014	0002AEC0	0013	00041AEE	FltBuildDefaultSecurityDescriptor
00000015	00039D50	0014	00041B10	FltCancelFileOpen
00000016	0000AE10	0015	00041B22	FltCancelIo
00000017	00029F50	0016	00041B2E	FltCancellableWaitForMultipleO...
00000018	00029F90	0017	00041B53	FltCancellableWaitForSingleObj...
00000019	0000AD20	0018	00041B75	FltCbdqDisable

图 18 - 27　fltMgr. sys 模块的导出函数

FltMgr 驱动框架允许注册多个小过滤驱动程序，也允许对同一种 I/O 操作注册多个过滤驱动。在一般性过滤框架中，过滤驱动的设备对象总是堆叠在设备栈的顶端，而且后加载的总是堆叠在先加载的驱动设备之上，因而 IRP/快速调用的下行方向总是后注册的先被过滤，上行方向总是后被过滤，限制了过滤拦截的灵活性。但在 FltMgr 框架中，FltMgr 允许由小过滤驱动自己指定位置和过滤顺序：通过高位值来决定小过滤驱动在驱动栈中的层次，值越大层次位置越高；值越小层次位置就越低，这样就非常方便地使能了过滤的层次指定。

图 18 - 28 展示了一个指定了高位值的小过滤驱动的例子。我们注册了 4 个小过滤驱

动，其中 A、B 两个驱动更接近 I/O 管理器，层次位置更高；C、D 两个驱动更接近文件系统驱动，层次位置更低(可以参考它们的高位值)。这些小过滤驱动有的负责防病毒，有的负责数据加密，它们都没有功能上的耦合。

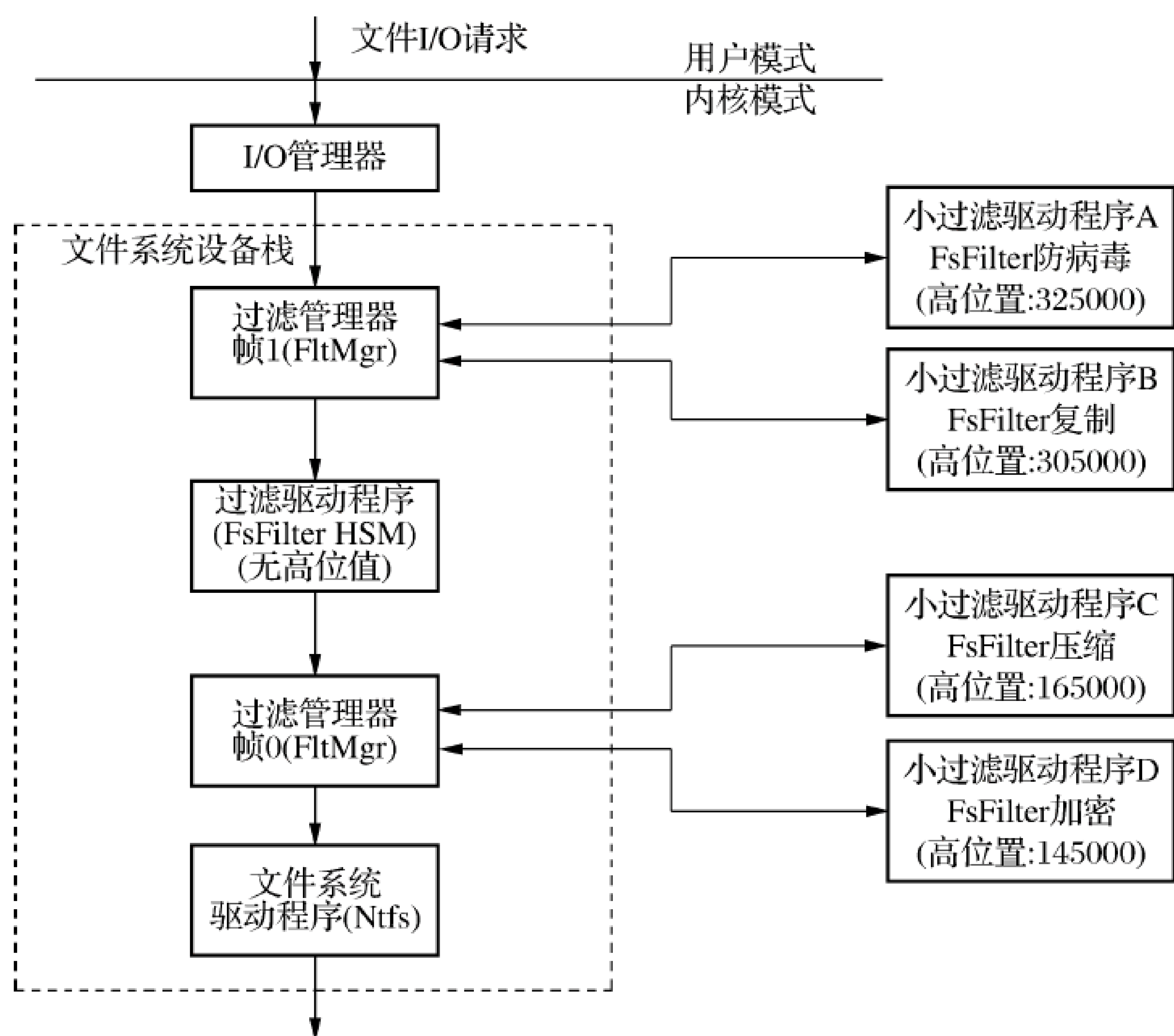


图 18－28　FltMgr 框架下的小过滤驱动

FsFilter HSM 是个传统的过滤驱动程序，包括了执行分层存储管理的筛选器驱动程序。虽然没有定义高位值，但实际上 Windows 为其定义的高位值恰好在图 18－28 的 A、B 和 C、D 之间。对于文件系统过滤器而言，HSM 之上的称为过滤管理器帧 1，HSM 之下的称为过滤管理器帧 0。

从图 18－28 还可以看出，小过滤驱动并不是通过设备堆叠的方式过滤 I/O 操作的，它自己也不会产生设备对象，它是包裹在 FltMgr 框架内部的，FltMgr 本身只有一个设备对象。

小过滤驱动通过 FltMgr 模块导出的 FltRegisterFilter 方法将自身注册到全局小过滤驱动链表中。注册时，小过滤驱动提供一组回调函数指针，当发生感兴趣的 I/O 操作事件时，FltMgr 框架可以对相应的函数进行回调。小过滤驱动通过 FltMgr 模块导出的 FltStartFiltering 方法通知 FltMgr 框架自己已经完成了各项准备工作，可以开始过滤拦截操作了。小过滤驱动一般通过入口函数初始化一些参数，然后调用 FltRegisterFilter 方法进行注册，最后调用 FltStartFiltering 方法通知开始过滤。

综上所述，文件系统小过滤驱动有如下优势：

- 对过滤节点的过滤顺序有更好的控制；
- FltMgr 框架支持小过滤驱动在运行时卸载；
- 可以只过滤选定的 I/O 操作；

➢ 不需要设备堆叠,不增加设备栈深度;
➢ 更友好的实现方式和更低的开发门槛;
➢ 扩展性更好。

本章小结

本章介绍了 Windows 文件系统。在介绍文件系统之前必须首先铺垫磁盘的相关技术和概念,因为文件系统的目标操作对象是磁盘上的扇区。

Windows 系统中是按照文件系统、卷、磁盘三层架构来构筑存储驱动协议栈的,其中卷是连接文件系统和磁盘驱动的中介,因此卷设备对象是需要重点介绍的内容。在此基础上也介绍了 FAT32、NTFS 等主流文件系统。

Windows 也支持对存储 I/O 进行过滤,FltMgr 就是一种新的文件系统过滤框架。

第19章 WDDM 框架

WDDM(Windows Display Driver Model,Windows 显示驱动模型)是微软新一代图形驱动模型框架,主要应用于 Windows Vista 及以上的版本,新版本的显卡驱动都需要遵守 WDDM 框架。WDDM 是符合 WDM 规范的小端口驱动,因而也具有电源管理和即插即用功能。

不同于 Windows XP 版本中的 XPDM(Windows XP Display Driver Model,Windows XP 显示驱动模型),WDDM 是对 XPDM 的改进,使用了 3D 加速技术,可以支持同时运行多个 GPU 密集型应用,而 XPDM 使用的是 2D 的 GDI(Graphics Device Interface,图形设备接口)或 GDI+(GDI 扩展,提供了诸如位图操作、画笔、抗锯齿和复杂的 Primitive 渲染中不同像素深度格式的支持),因此 WDDM 具有更好的性能和渲染效果。在 WDDM 框架下,所有的应用程序生成的显示画面都会在桌面窗口管理器(Desktop Windows Manager,DWM)内合成为单一的输出画面,并由此获得了更好的显示效果。

WDDM 框架大致可分成两个组件:内核模式驱动(KMD)和用户模式驱动(UMD)。WDDM 将大部分功能代码(主要是密集计算任务代码)从内核态移出到用户态,其中渲染显示相关的功能代码只在用户态下运行,从而提高了系统稳定性。

19.1 WDDM 框架基础库

WDDM 框架包含了三个 Windows 基础图形库:GDI、DirectX 和 OpenGL。

1. GDI

GDI(Graphics Device Interface,图形设备接口)是供普通用户使用的,用于绝大多数普通的二维图形界面的显示。GDI 通过调用 win32k.sys 模块的支持函数来模拟画图。win32k.sys 本质上是一个内核态的动态库,其并不是通过调用 IoCallDrivery 传递 IRP,而是通过导出名称带有"Eng"前缀的接口函数供 GDI/GDI+调用的方式来传递请求,如图 19-1 所示。

win32k.sys

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	002FBA74	002FC0FE	002FBE4C	002FCC07
(nFunctions)	Dword	Word	Dword	szAnsi
00000037	00198C8C	0036	0030FD9E	EngDeleteSemaphore
00000038	001A0D58	0037	0030FDB1	EngDeleteSurface
00000039	001C9168	0038	0030FDC2	EngDeleteWnd
0000003A	001994FC	0039	0030FDCF	EngDeviceIoControl
0000003B	001CB1A0	003A	0030FDE2	EngDitherColor
0000003C	0019AEF8	003B	0030FDF1	EngDxIoctl
0000003D	001CB348	003C	0030FDFC	EngEnumForms
0000003E	001A9E24	003D	0030FE09	EngEqualRgn
0000003F	001A117C	003E	0030FE15	EngEraseSurface
00000040	001CB3F8	003F	0030FE25	EngFileIoControl

图 19-1 win32k.sys 中用于绘图的接口

GDI 应用程序的图形界面先经过 CPU 处理后输出到主存上，再从主存传输到显卡驱动分配的 GDI 显存上（也是从主存的一块区域映射到显存中，是典型的“一块内存两次映射”，因而不采用 PCI-E 总线传输，效率较高），最后由桌面窗口管理器将画面合成输出到显示器上。

图 19－2 显示了 GDI 应用程序的各种窗口句柄对象。

21808	JisuPdf.exe	C:\Program Files (x86)...	X270-PC\tanzhe	0	0	8
23728	devicetree.exe	C:\Users\ZNV\Deskto...	X270-PC\tanzhe	0	0	3
25304	taskhost.exe	C:\Windows\system32...	X270-PC\tanzhe	0	0	0

Handle	Object Type	Kernel Address	Extended Information	Detect Counter	Detected On
0x1e0a12cc	Font	0xfffff900c2c21d...		1	2019/7/6 22:51:04
0x49043100	Region	0xfffff900c432c270		1	2019/7/6 22:51:04
0x4a103f1b	Brush	0xfffff900c46c7470		1	2019/7/6 22:51:04
0x6e0a2acf	Font	0xfffff900c23caa...		1	2019/7/6 22:51:04
0x80053f2d	Bitmap	0xfffff900c2cba0...		1	2019/7/6 22:51:04
0x8d0a38cd	Font	0xfffff900c2a4da...		1	2019/7/6 22:51:04
0x91013f9d	DC	0xfffff900c2efa630		1	2019/7/6 22:51:04
0xa3053d0b	Bitmap	0xfffff900c2ffd9a0		1	2019/7/6 22:51:04
0xa3103ad7	Brush	0xfffff900c69a83...		1	2019/7/6 22:51:04
0xb310103b	Brush	0xfffff900c46d69...		1	2019/7/6 22:51:04
0xb50a2864	Font	0xfffff900c1ff9780		1	2019/7/6 22:51:04
0xe00139aa	DC	0xfffff900c42eb0...		1	2019/7/6 22:51:04
0xf1013d03	DC	0xfffff900c1ff7630		1	2019/7/6 22:51:04

图 19－2　devicetree. exe 所占用的 GDI 资源

2．DirectX

DirectX（Direct eXtension）是由微软提供的多媒体编程接口，用于在 Windows 平台上的游戏或者多媒体进程的场景中获得更高的执行效率、更好的 3D 显示效果和音频效果，并为 Windows 程序提供高性能的多媒体硬件加速支持，如图 19－3 所示。

DirectX 接口 API 由 4 部分构成：显示部分、音频部分、输入部分和网络部分。

- **显示部分 API**：负责图像处理功能的 API，分为 DirectDraw（DDraw）和 Direct3D（D3D）两类，前者负责 2D 图像（例如视频播放、图片播放、2D 游戏等）加速，是对 GDI 的一种增强；后者负责 3D 效果（例如 3D 游戏中的人物）的展示。
- **音频部分 API**：负责播放声音、处理混音、录音和 3D 音效等功能的 API，最主要的 API 是 DirectSound 方法。
- **输入部分 API**：支持多种外设的输入，例如游戏输入设备、手柄、模拟器等。
- **网络部分 API**：支持包括 TCP/IP、IPX、串口等多种方式在内的 API，主要用于具有网络功能的游戏开发。

与 GDI 相比，DirectX 主要是针对 GPU 而设计的，面向的是专门的 3D 可视化计算任务及其加速效果，对于计算资源的消耗也主要集中在 GPU 及其相关硬件；而 GDI 瞄准的是普通的 2D 操作，动用的也是 CPU 的计算资源。因此从普适性上来说，GDI 强于 DirectX，但 DirectX 本身就是瞄准 3D 渲染加速的，其应用场景需要专门的 GPU 硬件支持也在情理之中。

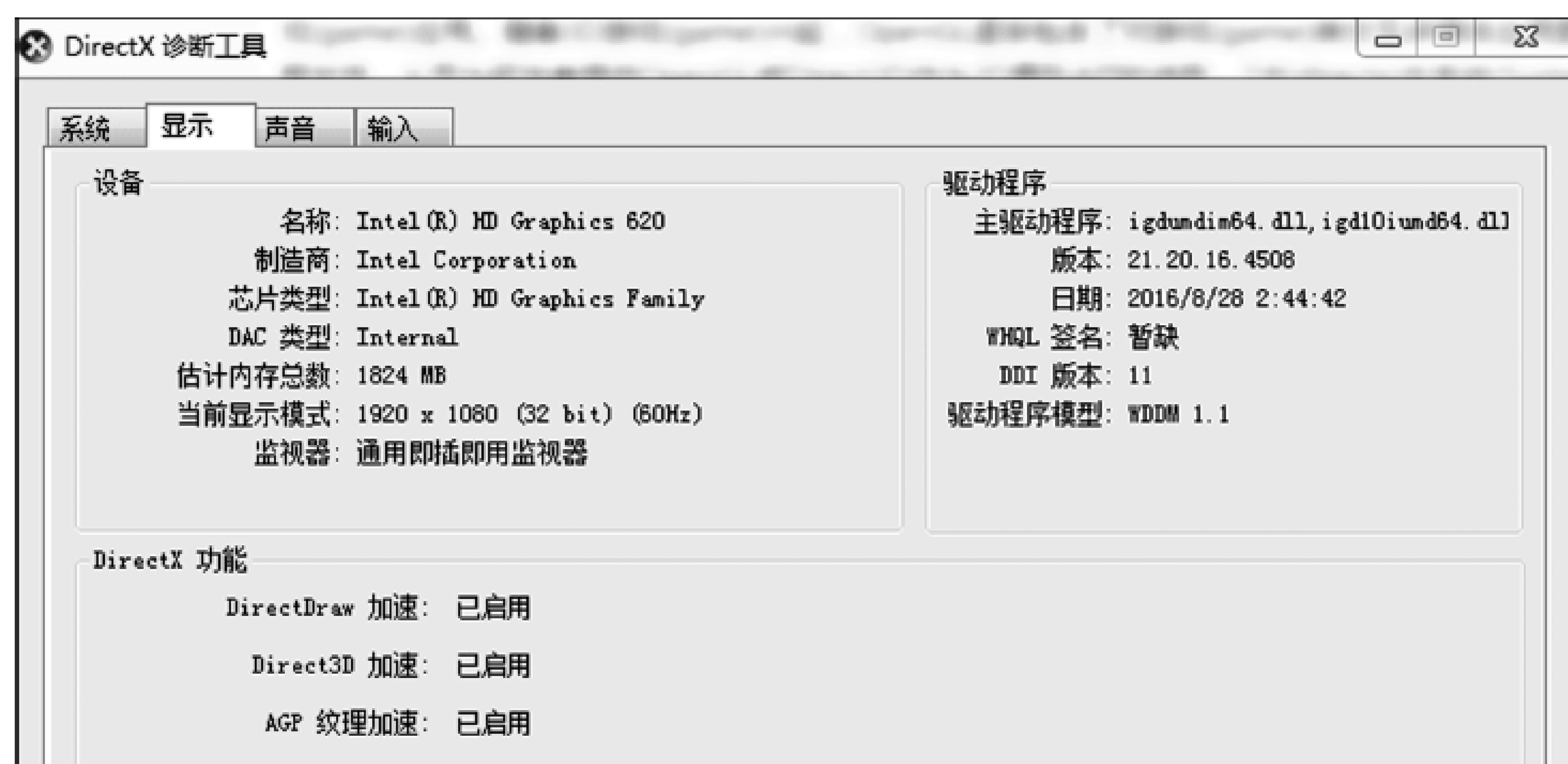


图 19－3　DxDiag 显示的 DirectX 信息

在 WDDM 框架中，DirectX 驱动也分成两部分：Windows 实现了在内核层的底层硬件相关的部分，DirectX 则实现用户层的软件相关的部分。

3. OpenGL

OpenGL（Open Graphics Library，开放图形库）是一套用于渲染 2D、3D 矢量图形的跨语言、跨平台且硬件无关的 API。其 API 总数接近 400 个，多用于 3D 可视化显示、游戏开发、虚拟现实、人工智能等场景。OpenGL 具有以下特性：

- OpenGL 通过一系列的几何图元（点、线段、三角形以及 patch）来创建三维空间的物体。
- OpenGL 是面向过程而不是面向对象的，不支持面向对象的编程思想。
- OpenGL 既可以软件实现，也可以硬件实现。图 19－4 中左边为软件实现，“构造图形”部分即为软件实现的模块，其下游为图形设备接口；右边为硬件实现，OpenGL 直接调用显卡驱动，显卡驱动则直接与显示设备打交道，并不经过图形设备接口。

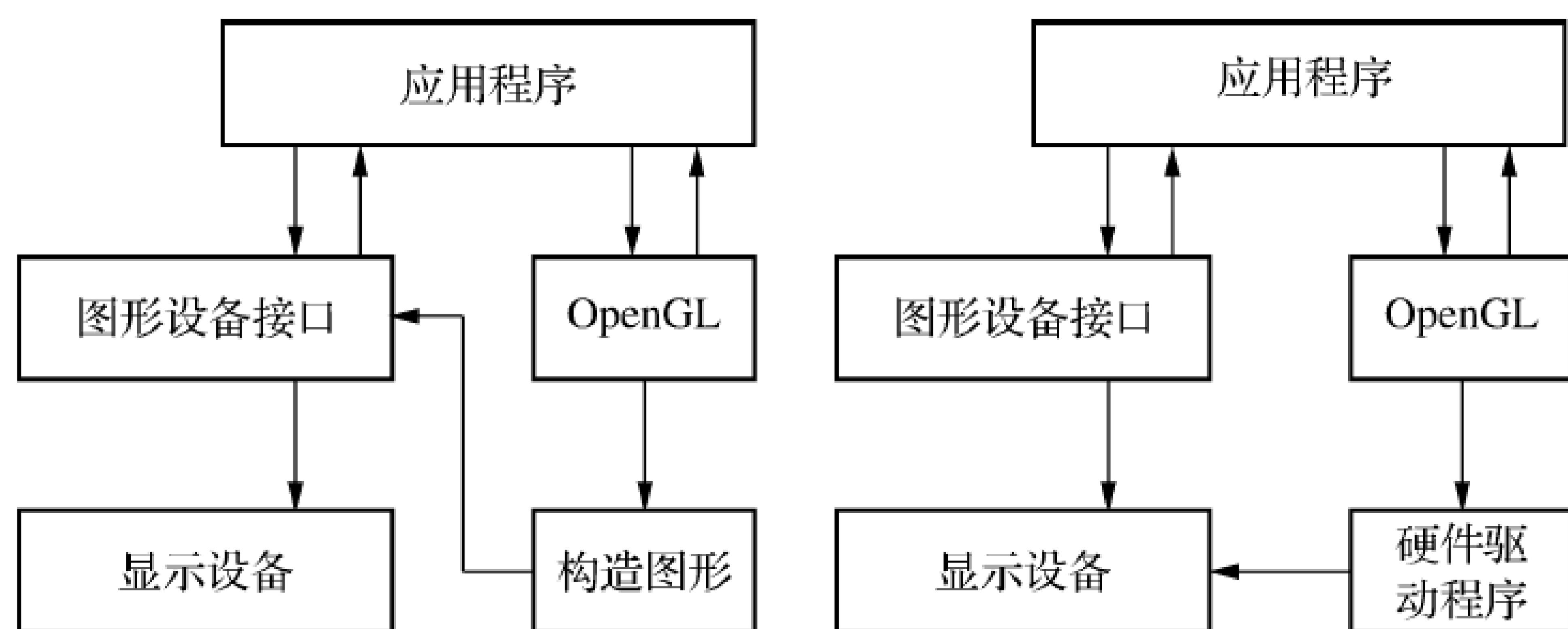


图 19－4　OpenGL 的软硬件实现

- OpenGL 采用了 C/S 设计模式，其客户端是使用 OpenGL API 的用户态进程，其服务端则是使用图形硬件厂商所提供的 OpenGL 实现。
- 在没有 OpenGL 时，CPU 与 GPU 之间的数据交换是通过内存控制器实现的，速率较低，而 OpenGL 创建了连续 RAM 的缓存区域以支持 CPU 与 GPU 之间高效率交换数据。

➢ OpenGL ES 是 OpenGL 的裁剪版本，专用于移动端的图像开发。

➢ OpenGL 基于渲染管线（pipeline）模型绘制图形。图形在应用进程中通过描述空间位置或顶点来指定其形状，这些顶点在处理过程中会流经 OpenGL 一系列模块，每个模块在图元（图形的基本组成部分）经过时对其实施一种或多种操作变换（例如旋转、平移、缩放等）。

OpenGL 导出的函数一般存在于两个库中，即 GL（OpenGL 核心库）和 GUL（OpenGL 实用库），前者包含了 OpenGL 所必需的函数，后者包含了新扩充的函数。总体来看，OpenGL 的导出函数分为以下几类：

➢ **图元函数**：用于指定要生成图形的图元，包括绘制二维/三维的集合图元的函数和绘制离散型实体（例如位图）的函数。

➢ **属性函数**：用于控制图元的外观及样式，例如颜色、文理、光照等。

➢ **观察函数**：用于对相机属性的操作，例如图像拉近、拉远的效果等。

➢ **控制函数**：用于启用 OpenGL 各种特性。

➢ **查询函数**：用于查询 OpenGL 各种状态变量。

上述三个基础图形库中，DirectX 和 OpenGL 需要在用户态空间中提供驱动模块以渲染图像，而相应的显卡驱动则处于内核态空间中。显卡驱动实际上是个即时编译器（Just-In-Time Compiler，JIT），会将图像相关的 API 所用的指令编译成 GPU 指令，同时负责在 CPU 和 GPU 之间传递图像。

在理解了上述三个 Windows 基础图形库后我们来看 WDDM 框架，如图 19－5 所示。

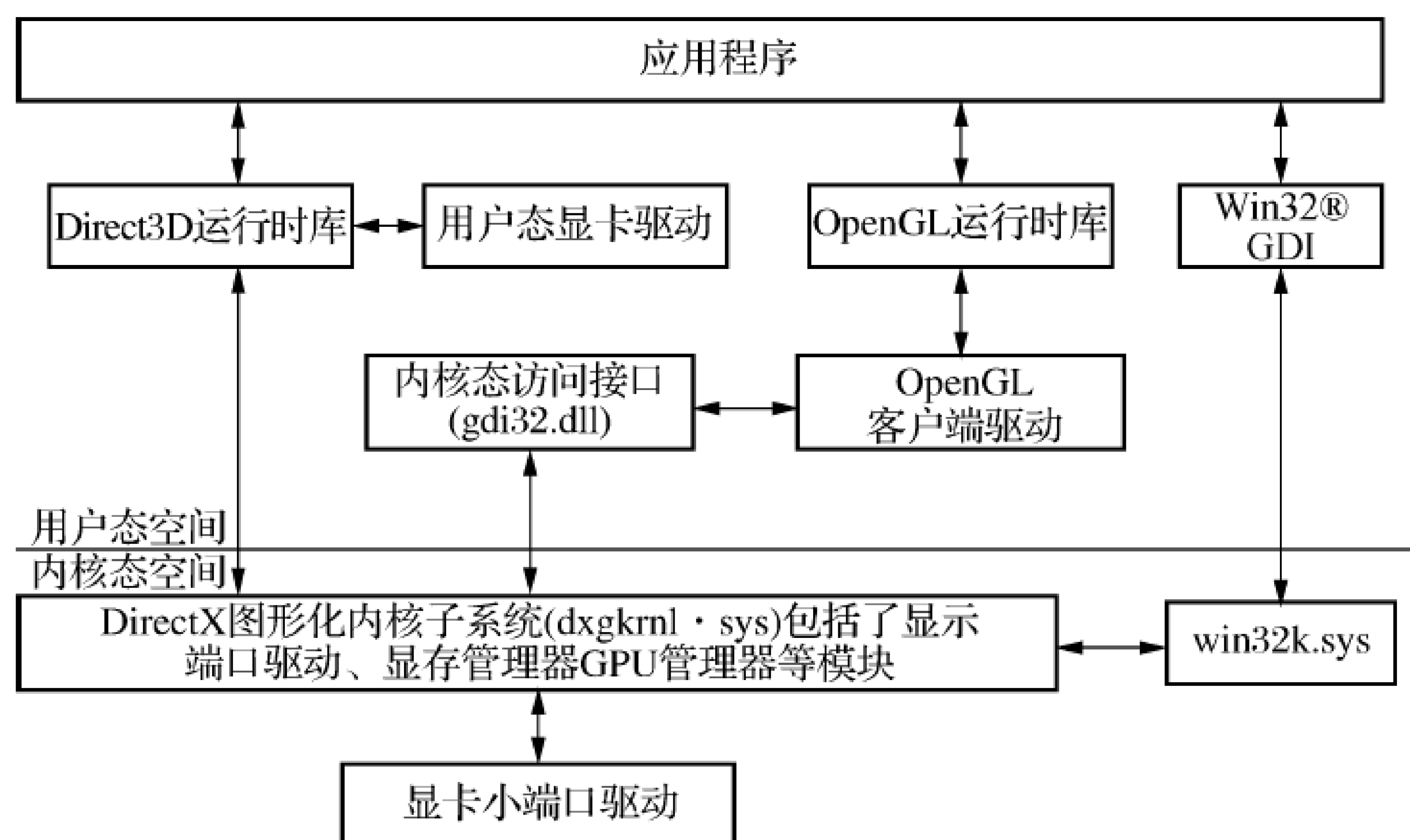


图 19－5　WDDM 框架视图

在 WDDM 框架中，应用进程通过上述三个基础库与显卡驱动打交道。其中：

- GDI 是与模块 win32k. sys 交互的，win32k. sys 再与 DirectX 图形内核子系统模块 dxgkrnl. sys 交互，并由 dxgkrnl. sys 统一向显卡的小端口驱动发送请求和指令。
- DirectX 通过 Direct3D 运行时库向应用进程提供支持，运行时库的大部分支撑功能是

在其用户态驱动程序中的，但也有一部分指令是直接发到 dxgkrnl. sys 中的。

- OpenGL 与 DirectX 一样，也拥有用户态的客户端驱动，也是通过 OpenGL 运行时库向应用进程提供支持的，但是 OpenGL 运行时库不直接与 dxgkrnl. sys 打交道，而是通过客户端驱动（为 OpenGL 接口服务的 OpenGL ICD 驱动）和 gdi32. dll 模块（是 Windows 下图形用户界面的应用拓展）发起各种请求的。

从图 19－6 可以看出，最终的图像都会通过 dxgkrnl. sys 驱动模块交互到内核模式的显卡驱动中。dxgkrnl. sys 作为 DirectX 图形内核子系统模块，提供以下功能：

➢ 管理显卡小端口驱动；

➢ 向上层的 DirectX、OpenGL 等提供通信支持；

➢ 作为适配层屏蔽下层各个厂商的显卡驱动的差异性，提供类似 NDIS 的回调接口。

dxgkrnl.sys

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	000E9A48	000E9B08	000E9A94	000E9BE2
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	0005CD28	0000	000EE138	TdrCompleteRecoveryContext
00000002	0005CC40	0001	000EE153	TdrCreateRecoveryContext
00000003	0005CEF8	0002	000EE16C	TdrIsRecoveryRequired
00000004	0005C68C	0003	000EE182	TdrIsTimeoutForcedFlip
00000005	0005D17C	0004	000EE199	TdrResetFromTimeout
00000006	0001F8E0	0005	000EE1AD	g_TdrConfig
00000007	0001F8F8	0006	000EE1B9	g_TdrForceTimeout
00000008	0000882C	0007	000EE1CB	DpSynchronizeExecution
00000009	0007FFA0	0008	000EE1E2	DpiGetDriverVersion
0000000A	00008E58	0009	000EE1F6	DpiGetDxgAdapter

图 19－6　dxgkrnl. sys 的导出函数

内核模式显卡驱动的作用则包括资源的分配管理、主存与显存之间的数据交换（DMA 方式）以及 GPU 管理等，这些都是与具体显卡相关的管理功能了。

WDDM 上层的用户模式驱动用于支持 Windows 基础图形库的计算任务，按照功能大致包括了以下组件：

➢ **Display 组件**：负责渲染和显示，包括分辨率处理、刷新率设置、多屏显示等。

➢ **2D 组件**：负责画点画线、内存快速拷贝等 2D 功能，目前已经比较少用，多数 2D 功能也可采用 3D 组件来完成。

➢ **3D 组件**：负责 3D 图像的显示和加速任务。

➢ **Video 组件**：负责视频编解码的硬件加速。

➢ **通用计算（General Purpose Computing）组件**：负责对 OpenCL、OpenCV、CUDA（Compute Unified Device Architecture，统一计算设备架构）等图像处理框架的支持。

这些图像处理框架一般通过两种方式操控显卡，一种是使用内核系统调用机制，包括申请、映射、释放显存对象和执行 GPU 指令等；另一种是在用户态驱动就把 API 翻译成 GPU 指

令，再通过内核态驱动将这些指令下达给 GPU。

19.2 WDDM 框架与显卡驱动的交互

Dxgkrnl 作为 DirectX 图形内核子系统模块对下屏蔽了各种类型显卡的差异性。Dxgkrnl 借鉴了 NDIS 的设计思想，向下层设备驱动提供回调函数供显卡驱动调用。

显卡驱动的入口函数依然是 DirverEntry。在 WDDM 框架下，任何厂商的任何型号的显卡驱动都要遵循 WDDM 框架的规则：

- 入口函数中需要调用 WDDM 框架提供的 DxgkInitialize 方法，用于注册显卡驱动回调函数集数据结构 DRIVER_INITIALIZATION_DATA，这个数据结构中包括了显卡回调函数指针，例如 DxgkDdiAddDevice、DxgkDdiStartDevice 等；也包括图像相关的 DDI（设备驱动接口）函数，例如图像资源分配接口等；还包括了管理 VIDPN 的函数。调用该方法完成后，显卡驱动的回调函数就注册到了 dxgkrnl. sys 中。
- 上述注册完成后，WDDM 框架回调显卡驱动注册的 dxgkDdiStartDevice 方法，并将数据结构 PDXGKRNL_INTERFACE 作为参数传递进去。该数据结构承载了 dxgkrnl. sys 提供给显卡驱动的回调函数指针以便于下层驱动调用。

至此，显卡驱动和 WDDM 框架完成了各自的回调函数指针的互相持有。

在这里我们要解释一下 VIDPN。VIDPN 即视频呈现网络（Video Present Network），这是一个抽象的概念，表示显卡的 Source 和 Target 之间的连接关系，即哪些 Source 分别向哪些 Target 输出图像，如图 19－7 所示。

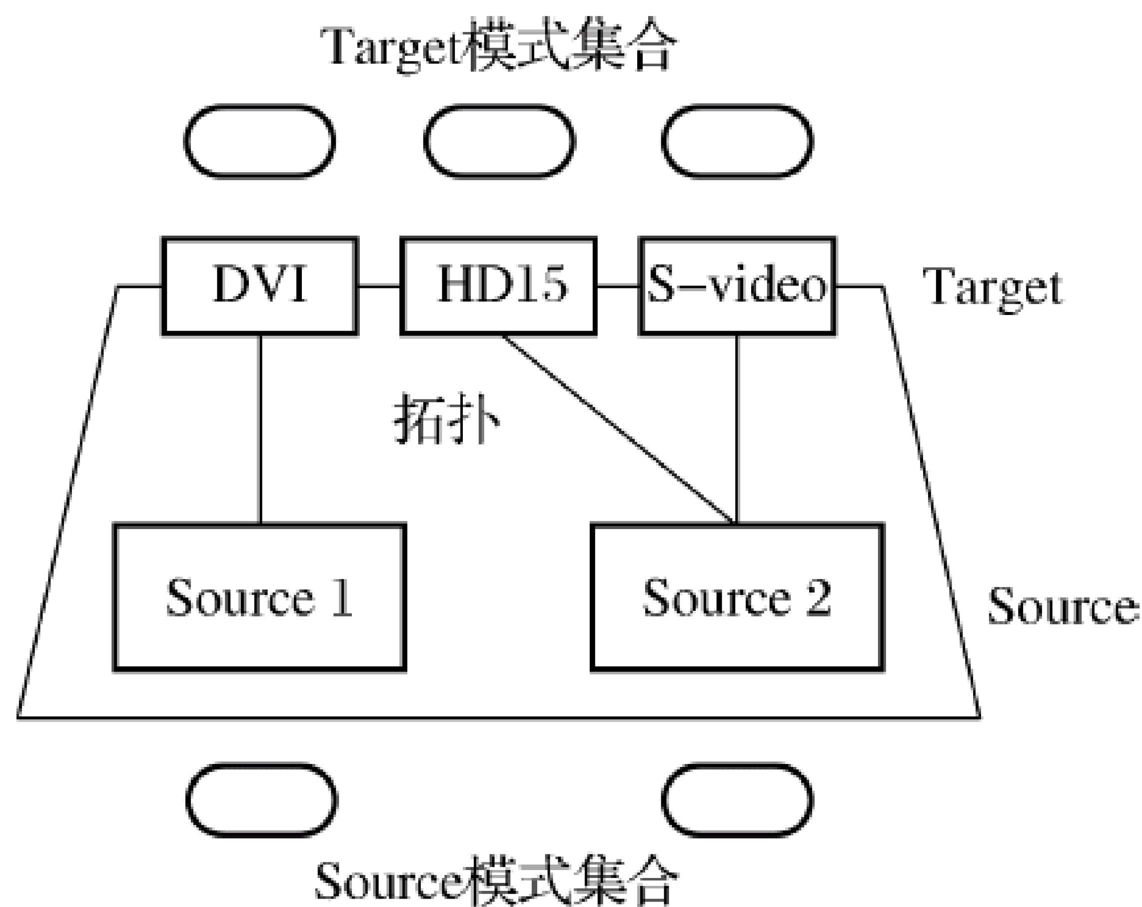


图 19－7 Source 与 Target 之间的连接关系

其中，Source 是显卡的显示源，即图像的输入侧，代表了显卡表面的 Surface，所有的桌面图像都会画到这个 Surface。显卡一般都有两个或两个以上的 Source，一个作为主显示表面，即 Windows 桌面；另一个则作为扩展桌面显示源，例如一机双屏的显示方式。Target 则表示显示接口，一般包括 VGA 接口、HDMI 接口、Display-Port 接口以及其他种类的输出接口。显示源 Source 都会向某个或若干个显示接口 Target 以输出最终的图像。

无论是 Source 还是 Target 都有自己的模式集合，这些模式包括分辨率、颜色深度等。Source 的路数受制于显卡，但 Target 的数量可以通过软件的方式任意扩展。

结合 DirectX，WDDM 的画图操作执行流程如图 19－8 所示，这里我们只介绍其中的几个步骤：

（1）应用程序请求创建渲染设备，DirectX 运行时库向内核态的 dxgkrnl. sys 发送创建请

求,dxgkrnl.sys 调用方法 DxgkDdiCreateDevice 创建设备对象 DXGKARG_CREATEDEVICE,该设备对象包含了 DMA 的有关信息。

(2) DirectX 运行时库调用 DirectX 的用户态驱动的 CreateDevice 方法以创建一个图形上下文 D3DDDIARG_CREATEDEVICE。在 CreateDevice 的调用中,用户态驱动需要调用 pfnCreateContextCb 函数为新建的设备创建一个或多个 GPU 执行线程。

(3) DirectX 运行时库调用其用户态驱动的资源创建函数 CreateResource,在该函数的执行中应调用 pfnAllocateCb 函数。

(4) 内核态显卡驱动接收到 DxgkDdiCreateAllocation 调用,调用中包含了为设备分配的资源的数量和类型。函数返回的分配信息在 DXGKARG_CREATEALLOCATION 结构体的成员变量 pAllocationInfo 指针列表中,pAllocationInfo 是个 DXGK_ALLOCATIONINFO 数据结构。

(5) 应用程序发出一个画图请求后,Direct3D 会调用用户态显卡驱动相关的画图操作。

(6) 用户态显卡驱动会与内核态显卡驱动交互,最终完成图像输出。

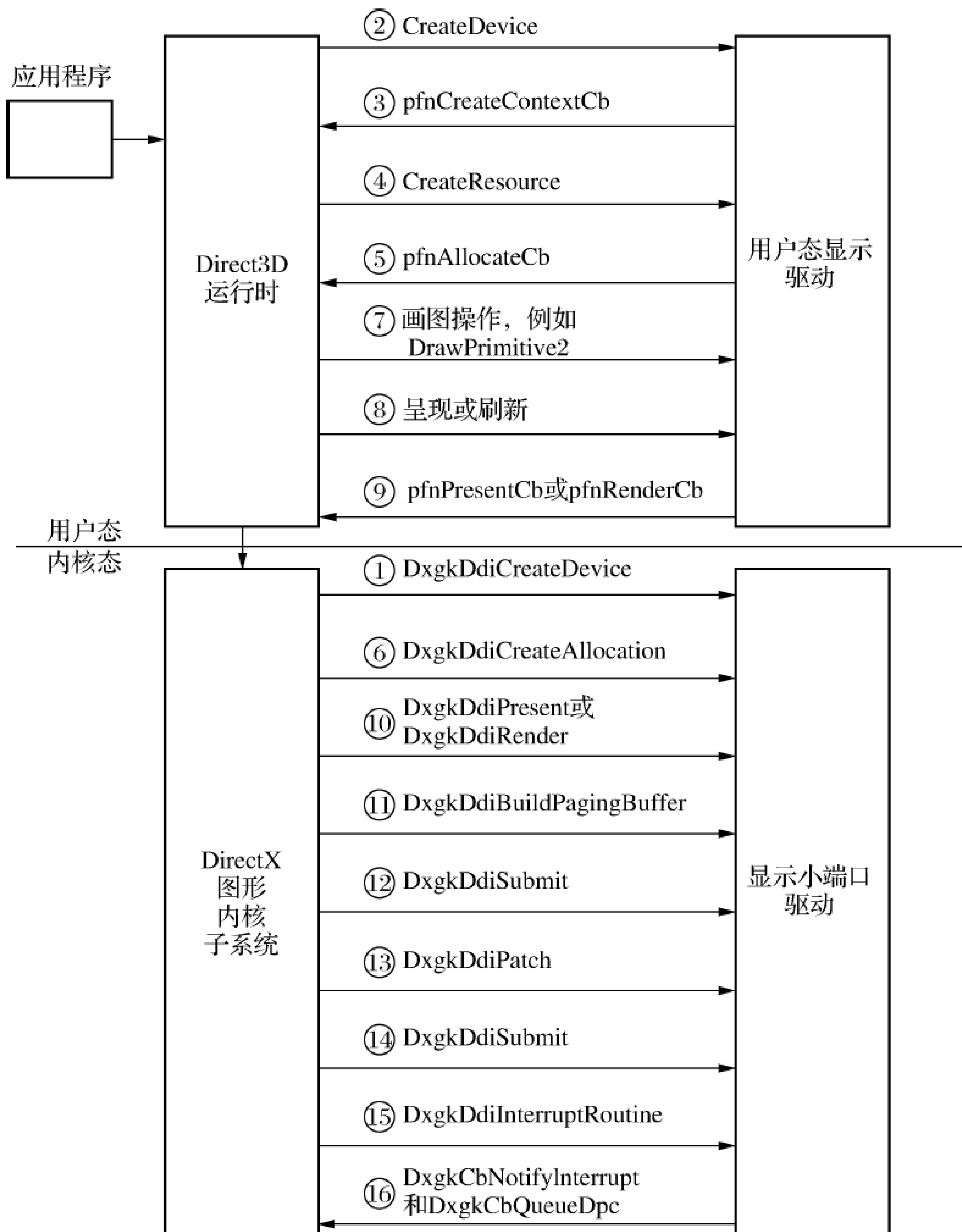


图 19－8　WDDM 框架的画图操作执行流程

19.3 虚拟化桌面的显示支持

虚拟化桌面(云桌面)与普通的台式机桌面(普通桌面)有很大的不同,当然这不是说桌面呈现风格的不同,而是指桌面显示技术的不同,即通过远程瘦客户端查看到的云桌面图像和在普通 PC 显示器上看到的本地操作系统界面图像,两者的传输方式是不同的。图 19-9 是云桌面图像和普通桌面图像的显示框架。

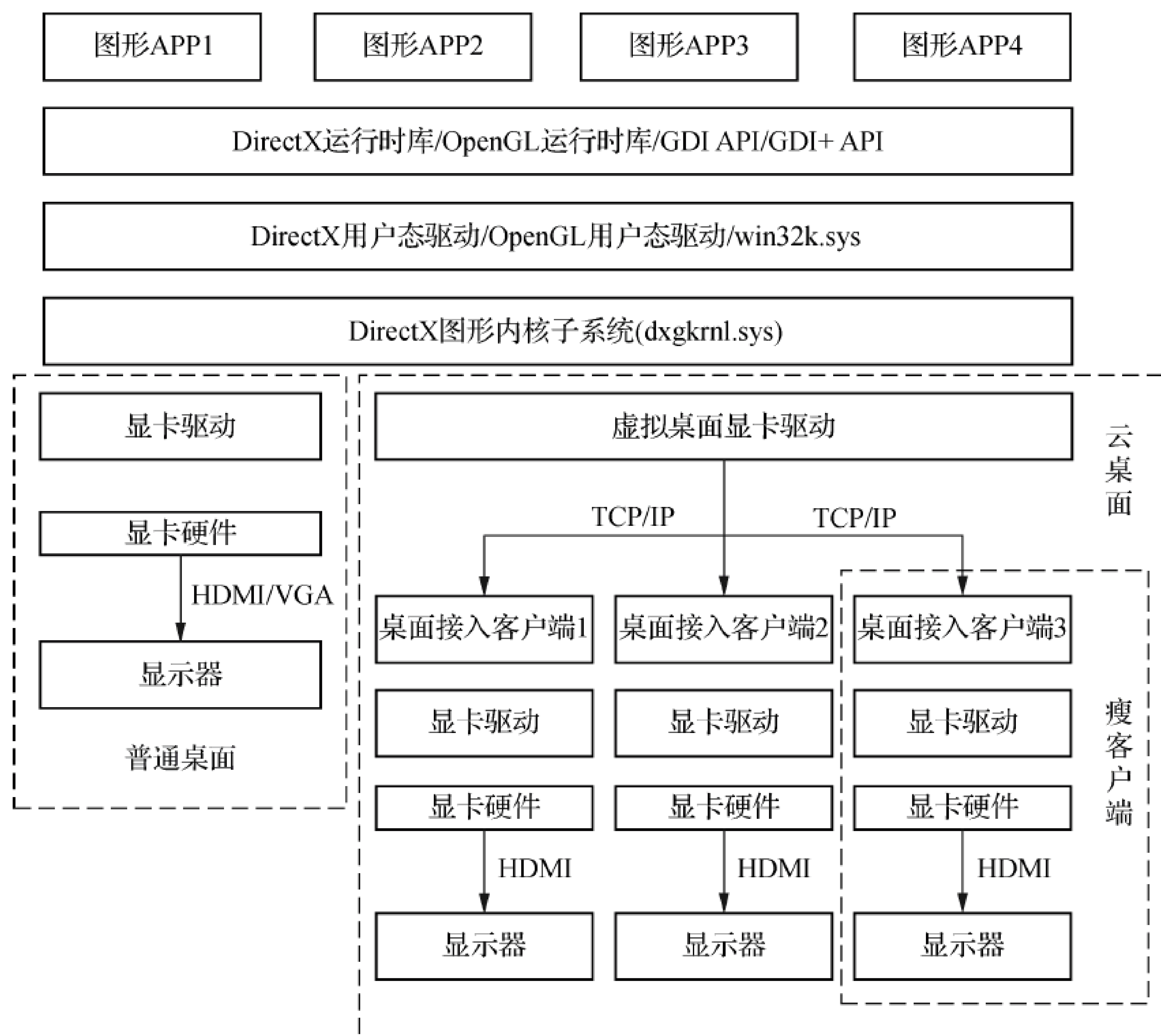


图 19-9　云桌面与普通桌面的显示框架

当前的云桌面操作系统基本为 Windows,因此以 Windows 为例来讲述云桌面的显示技术比较有代表性。

从图 19-9 可知,云桌面与普通桌面的显示框架的不同主要在下层,即图形内核子系统之下。

- 普通桌面系统通过图形内核子系统调用显卡驱动,显卡驱动进行图像的渲染和加速并通过显卡设备输出到显示器。
- 云桌面系统中,服务器端代替硬件显卡驱动的是虚拟桌面显卡驱动,从图形内核子系统(dxgkrnl. sys)的角度看,这是显卡小端口驱动,但本质上这是一个编码器,即将云桌面的图像进行压缩,并通过 TCP/IP 协议传输到瘦客户端的接入管理程序中。作为

瘦客户端,其最接近 TCP/IP 的一层是桌面接入客户端,其任务就是接收远程的桌面压缩信号并传递到瘦客户端的显卡驱动中,显卡驱动解码渲染后将桌面信号内容发送至显卡硬件,继而转换成 HDMI 信号输出到显示器中。

当然,通过 TCP/IP 协议传输桌面图像的带宽压力是很大的。因此云桌面系统中大多运行 2D 进程而较少运行计算量大的 3D 进程,而且对于图形指令和图像内容进行了各种优化,例如图像数据压缩(有损和无损压缩)、指令合并以及缓存技术等。

本章小结

本章总结了 Windows 图形框架的有关细节,包括基础库、与显卡驱动的交互过程以及虚拟桌面的显示技术等。

第20章　Rootkit 技术

Rootkit 是一种特殊的恶意软件,其功能是在目标系统中隐藏自身,同时获取目标系统中软件的相关信息并发送给恶意软件控制方,或者在目标系统中进行破坏行为。可以看出,Rootkit 就是指病毒、高级可持续威胁(Advanced Persistent Threat,APT)、木马等恶意代码或软件。Rootkit 所使用的技术也包括挂钩技术、驱动过滤技术等,甚至包括更为底层的 Bootkit 技术,而这些技术正是多年来病毒与反病毒、APT 与反 APT、入侵检测系统(Intrusion Detection Systems,IDS)、入侵防御系统(Intrusion Prevention System,IPS)、深度包检测(Deep Packet Inspection,DPI)等领域常用的基本技术,也是"必杀技"。Rootkit 涉及的操作系统不仅仅是 Windows、Linux、Android、iOS,甚至一些特殊领域的工控系统都在其涉猎范围。PC、服务器、虚拟机、公有云等环境也都是 Rootkit 生长的沃土。

可以说在这场旷日持久的浩大战争中,系统的攻击者与防御者之间的拉锯日益激烈,有些技术今天还是攻击方的王牌,明天可能就成了防御者的神盾。所谓技术没有善恶,没有攻击者魔高一尺的全面进攻,就没有防御者道高一丈的全线阻击。而作为防御者的友军,操作系统和设备驱动也在不断进化以提升自己的免疫和防御能力。"小成功靠朋友,大成功靠敌人",正是在这场无休无止的攻防战中,系统的攻击者和防御者都进化出了自己的独门绝技。

不过本章主要是讲述 Rootkit 相关技术本身,不涉及病毒与反病毒领域,更不涉及狭义的网络通信安全领域,后面会有专门的章节讲述这些领域的相关技术。

20.1　挂钩技术

挂钩技术,即 Hook 技术,这是个在操作系统、网络安全、软件逆向等领域大名鼎鼎的古老技术。所谓"挂钩",本质上就是在软件正常执行流程中"嵌入"一个楔子,使软件的执行流程发生转变。当然,这个"嵌入"的形态是多种多样的,例如 Inline Hook、IAT Hook、SSDT Hook、IDT Hook、APC Hook 等。挂钩技术在 Windows\Linux\Android\iOS 等平台上均有广泛的应用。

挂钩技术在软件调试、故障诊断、参数检测、程序破解等日常调试操作中具有重大作用。在调用非开源的动态库文件时,特别是在解决非开源第三方库的互相调用过程中的参数监测等问题时,如果采用 Inline Hook 或者 IAT Hook 的方式会非常直观地检测出参数的实际值,从而辅助软件调试和故障诊断。

同时,挂钩技术也是病毒与反病毒领域最基本的技术之一。几乎所有的病毒或 APT(高级可持续威胁)都采用了挂钩的思想,在操作系统内核、驱动、浏览器、应用程序等多种软件

体系内不断地寻找挂钩点,不断地进行着争夺与反争夺的战争。而在网络安全对抗领域,针对网络协议栈驱动的过滤与挂钩则是 IDS 和 IPS 的基本技术手段。

以上这些技术从广义来说都可以归类为挂钩技术。

20.1.1　Inline Hook

Inline Hook,即内联挂钩,是最原始、最底层的一种挂钩方式,其本质上就是通过改变目标函数代码头部的若干字节来进行流程截获和跳转的技术。例如在 32 位系统中将函数汇编代码中的 5 个字节替换成 jmp 0x12345678,则函数执行到这里的时候就会跳转到本进程空间内的 0x12345678 位置。这里的"字节"是二进制文件的机器码,而不是 C 语言或者 Java 语言的几个字节的代码。在 Windows 下,C/C++ 语言经过编译链接之后会形成 PE 文件,DLL、EXE、SYS、OCX 等格式的文件都是 PE 文件。采用 OllyDbg 等工具打开 PE 文件,看到的十六进制字节就是机器码,如图 20-1 所示。

```
10294AB9    7E BD            jle short 10294A78
10294ABB    FF0D 68954610    dec dword ptr [10469568]
10294AC1    390D 20954610    cmp dword ptr [10469520],ecx
10294AC7    75 05            jne short 10294ACE
10294AC9    E8 F2EFFFFF      call 10293AC0
10294ACE    E8 9E590000      call 1029A471
10294AD3    E8 88300000      call 10297B60
10294AD8    E8 1D420000      call 10298CFA
10294ADD    EB 0C            jmp short 10294AEB
10294ADF    83F8 03          cmp eax,3
10294AE2    75 07            jne short 10294AEB
10294AE4    51               push ecx
10294AE5    E8 0E310000      call 10297BF8
10294AEA    59               pop ecx
10294AEB    6A 01            push 1
10294AED    58               pop eax
10294AEE    C2 0C00          retn 0C
10294AF1    55               push ebp
10294AF2    8BEC             mov ebp,esp
10294AF4    53               push ebx
```

图 20-1　在 OllyDbg 中观察到的进程的汇编指令和机器指令

图 20-1 中的第二列为编译后的机器码,第三列为机器码对应的汇编指令,当然这个关系都是固定的,只跟 CPU 平台架构和平台编译器有关,同一个高级误言指令在 X86 和 ARM 架构下编译的结果是不一样的。例如在 X86 架构下,一条高级指令可以拆解为两条汇编指令:"push ebp"对应的机器码就是 0x55,"mov ebp,esp"对应的机器码是 0x8BEC;而在 ARM 架构下,可能同样一条高级指令对应的就是三条汇编指令,对应的机器码也不一样了。其实不要说在 X86 和 ARM 这样一个是复杂指令集、一个是精简指令集的两个架构中,就是在同属复杂指令集且同属 X86 体系的 X86 和 X64 架构中,其编译的结果也不尽相同(X64 编译器在处理 64 位程序时会有更多的寄存器可以选择,因此能不依赖堆栈就不依赖堆栈,这与 X86 寄存器少,大多数情况下需要依赖堆栈传参有很大不同)。

Inline Hook 的位置大多在函数头部,当然也可以在中间或尾部,这取决于我们的具体实现需求和挂钩的难易度。头部的指令相对简单和固定,当然这也是编译器留下的一个"后门",以备挂钩的不时之需。挂钩后可以接管函数的整个流程,不需要考虑函数内部业务逻



辑的割裂(在函数中汇编指令会把原始的高层语言指令拆得很细碎,一条高级语言指令可能会对应许多条汇编指令,从程序逻辑角度去分割对应的汇编语言指令的边界很困难),也不需要考虑压栈、出栈和堆栈配平。

Inline Hook 可以采用一个 jmp 指令跳转,其操作手段的门槛很低,不需要考虑太多的"包袱"。

Inline Hook 的大致流程如图 20-2 所示,其中左侧是函数执行的正常流程,右侧是挂钩后的流程。可以看到原来的执行流程被截断了,一个 jmp 指令使流程跳到了旁路指令序列以便完成挂钩的业务,之后再执行原来被跳过的指令序列 1,并最终执行完成原有的所有流程。这个流程非常简单直观,需要注意的是:

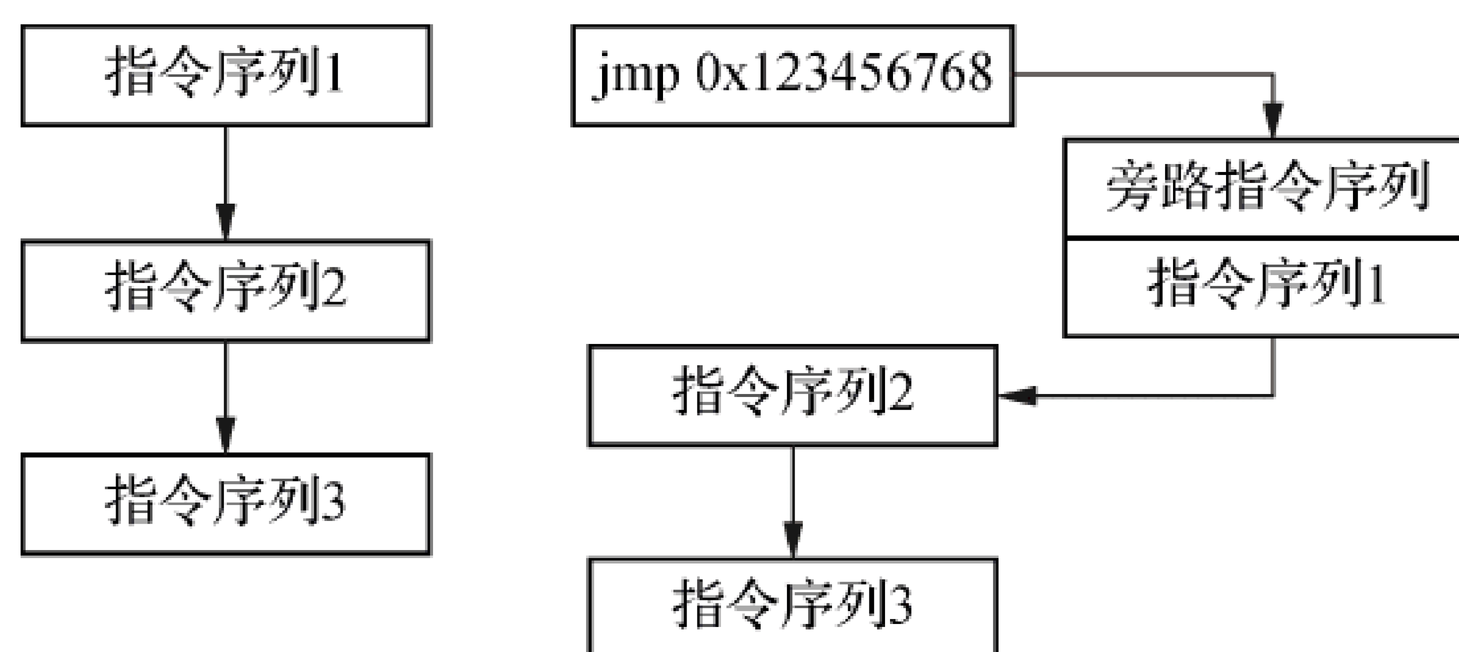


图 20-2 Inline Hook 的执行流程

- 原来的指令序列 1 需要被保存,因为在挂钩的业务逻辑执行完后还要回来继续完成原有的流程。
- 跳转指令一般是 5 个字节(32 位系统下),如果 Inline Hook 跳转指令占用的字节数未与原始的指令序列对齐(即没有正好占用一条汇编指令的大小),需要用 nop 空操作进行填充,以防止汇编指令产生歧义。
- 跳转指令不止 jmp 一种,还有 call 指令、push ret 指令对等。这些指令占用的字节数不同,可以进行多种选择,以便于对不同原始指令字节数的覆盖。

这里我们需要解释一下上述三个跳转指令(jmp、call、push ret)的区别。首先来看跳转的两种方式:

- **段内转移**:这种跳转方式不修改 CS 寄存器(段寄存器),因为不需要切换段,所以只需要修改 EIP 寄存器,例如 jmp eax。
- **段间转移**:这种跳转方式既要修改 CS 寄存器也要修改 EIP 寄存器,因为要进行段间跳转,所以 CS 寄存器的内容也会变化,例如 1000:0。

明白了两种跳转方式后,接下来看看上述三个跳转指令的异同。

1) jmp 指令

jmp 指令分为短跳转(Short Jump)、近跳转(Near Jump)和远跳转(Far Jump)3 种方式,短跳转和近跳转对应段内转移,远跳转对应段间转移。CPU 在执行 jmp 指令时并不需要转移的最终目的地址,而是只需包含转移的位移就可以了。这个位移是目的地址和当前 jmp 指令之间的差值。**jmp 指令既不影响堆栈,也不保存返回地址**。

2）call 指令

call 指令在进行流程跳转前会保存返回地址（call 指令的下一条指令的地址），以便在跳转的目标代码中使用 ret 指令返回到 call 指令的下一条指令处继续执行，这个返回地址的保存是要压栈的，因此 **call 指令既影响堆栈，也保存返回地址**。执行段内跳转（或称“near call”）时只保存 EIP 寄存器的值；如果是段间跳转（或称“far call”），还要保存 CS 寄存器的值。

3）push ret

ret 指令在返回时从堆栈中取得 EIP 指令（返回地址），因此在执行 ret 指令之前应该通过 push 指令将返回地址压栈。**push ret 指令既影响堆栈，也保存返回地址**。

在 Windows 系统中，有很多进行 Inline Hook 的点，我们将其称为“挂钩点”。内核中，特别是 SSDT、IDT、系统驱动模块（如 win32k. sys），它们都是挂钩的“重灾区”。因为在这些区域中挂钩更为底层，可以规避用户态进程隔离的问题。试想如果在用户态的单个进程内挂钩，虽然该进程内的系统调用活动可以被监控，但是对于其他进程就无法监控了，只能在其他进程中的相同点上再挂钩。另外，在一些公用模块中进行 Inline Hook 也是个好主意（例如 user32. dll 等公用库）。许多杀毒软件、系统防护进程都是在内核态空间进行挂钩，当然不仅仅限于 Inline Hook，还有其他形式的挂钩方式都可以在内核里实现。Inline Hook 是内核态和用户态都能使用的挂钩方式，但是随着检测技术的进步，Inline Hook 被检测和移除的可能性也空前增大。

Inline Hook 本质上是对内存模块的覆写，一般被覆写的内存页面都具有可执行权限。但对于 Windows 系统来讲，可执行内存区域一般是不能写的，因此需要通过 VirtualProtect 等方法改变当前内存页面的读写属性，使之可读、可写、可执行。覆写完成后依然要调用 VirtualProtect 方法恢复原来的读写权限。

那能不能对磁盘上的模块文件进行覆写，使之加载到内存时已经是内联挂钩过的呢？当然可以，但那就是文件替换或文件更改而不是内存 Inline Hook 的范畴了，我们在这里专门指的是运行时内存改写这种“热”方式。

图 20－3 和图 20－4 分别是 Windows 7 环境下的 ShadowSSDT 中的 Inline Hook 和 SSDT 中的 Inline Hook 实例。可以看到在 ShadowSSDT 中共有 7 个接口进行了 Inline Hook，SSDT 中进行 Inline Hook 的数量更多。这些都是在内核态空间进行的挂钩行为。

vowhrmudinjqa

进程 | 驱动模块 | 内核 | 内核钩子 | 应用层钩子 | 网络 | 注册表 | 文件 | 启动信息 | 系统杂项 | 电脑体检 | 配置 | 关于

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
3	NtUserGetKeyState	0xFFFFF960000F5F3C	inline hook	0xFFFFF960000F5F3C	C:\Windows\System32\win32k.sys
68	NtUserGetAsyncKeyState	0xFFFFF960000EE504	inline hook	0xFFFFF960000EE504	C:\Windows\System32\win32k.sys
120	NtUserGetKeyboardState	0xFFFFF960000F5DB8	inline hook	0xFFFFF960000F5DB8	C:\Windows\System32\win32k.sys
236	NtUserAttachThreadInput	0xFFFFF960000ED478	inline hook	0xFFFFF960000ED478	C:\Windows\System32\win32k.sys
701	NtUserGetRawInputBuffer	0xFFFFF960000FAFCC	inline hook	0xFFFFF960000FAFCC	C:\Windows\System32\win32k.sys
702	NtUserGetRawInputData	0xFFFFF960000FA730	inline hook	0xFFFFF960000FA730	C:\Windows\System32\win32k.sys
755	NtUserRegisterHotKey	0xFFFFF960000EF728	inline hook	0xFFFFF960000EF728	C:\Windows\System32\win32k.sys

图 20－3　Windows 7 系统中 ShadowSSDT 的 Inline Hook 实例

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

挂钩对象	挂钩位置	钩子类型	挂钩处当前值	挂钩处原始值
len(3) ExFreePool[ntoskrnl.exe]	[0xFFFFF8000483F000]->[-]	Inline	02 00 08	00 00 00
len(1) [ntoskrnl.exe]	[0xFFFFF800046F9372]->[-]	Inline	21	01
len(1) [ntoskrnl.exe]	[0xFFFFF800046F94FF]->[-]	Inline	29	09
len(1) [ntoskrnl.exe]	[0xFFFFF800046F96C2]->[-]	Inline	21	01
len(1) [ntoskrnl.exe]	[0xFFFFF800046F9851]->[-]	Inline	29	09
ntoskrnl.exe:KeUserModeCallback[win32k.sys<=...	[0xFFFFF800049373D0]->[0xFFFFF8800425E9...	Iat	2C E9 25 04 80 F8 F...	D0 73 93 04 00 F8 F...
len(14) [win32k.sys]	[0xFFFFF960000ED5E8]->[-]	Inline	FF 25 00 00 00 00 A...	48 89 5C 24 08 48 8...
len(14) [win32k.sys]	[0xFFFFF960000EE674]->[-]	Inline	FF 25 00 00 00 00 9...	48 89 5C 24 08 48 8...
len(14) [win32k.sys]	[0xFFFFF960000EF898]->[-]	Inline	FF 25 00 00 00 00 F...	48 8B C4 48 89 58 0...
len(14) [win32k.sys]	[0xFFFFF960000F5F28]->[-]	Inline	FF 25 00 00 00 00 4...	48 89 4C 24 08 53 5...
len(14) [win32k.sys]	[0xFFFFF960000F60AC]->[-]	Inline	FF 25 00 00 00 00 E...	48 89 5C 24 08 48 8...
len(14) [win32k.sys]	[0xFFFFF960000FA8A0]->[-]	Inline	FF 25 00 00 00 00 C...	48 8B C4 48 89 58 0...
len(14) [win32k.sys]	[0xFFFFF960000FB13C]->[-]	Inline	FF 25 00 00 00 00 6...	48 89 4C 24 08 53 5...
len(6) [srv.sys]	[0xFFFFF88003D05353]->[-]	Inline	15 F8 2C FB FF 90	94 C2 60 D7 01 00

图 20－4　Windows 7 系统中 SSDT 的 Inline Hook 实例

而为了监控用户态进程的行为,也经常在应用进程的公共模块中进行 Inline Hook,相当于在用户态层面拦截了它们的必经之路,这在 Windows 中也是常见的做法。在用户态进行挂钩更稳定,对系统的破坏更小。因为在内核态一旦不慎因挂钩而造成了错误,导致的是 BSOD 式的毁灭性后果,而用户态出现错误顶多就是当前进程挂掉而已。图 20－5 就是对用户态的公共模块 user32. dll 进行的 Inline Hook 实例。

映像名称	进程ID	父进程ID	用户名	进程路径	占用内存	运行时间	进程参数	进程类型
webservd.exe *32	2368	632	SYSTEM	C:\Program Files (x86)\I...	4M	12:33:48		这是一个
micmute.exe *32	2412	632	SYSTEM	C:\Program Files\Lenovo\...	7M	12:33:48		这是一个
tphkload.exe	2436	632	SYSTEM	C:\Program Files\Lenovo\...	11M	12:33:48		这是一个
SogouCloud.exe ...	2632	6400	tanzhe	C:\Program Files (x86)\S...	34M	12:20:56	-daemon	搜狗输入
Locator.exe	2640	632	NETWO...	C:\Windows\System32\L...	2M	12:33:47		这是一个
svchost.exe	2752	632	NETWO...	C:\Windows\System32\s...	5M	12:33:47	-k NetworkSe...	这是一个
Foxmail.exe *32	2848	3284	tanzhe	D:\Program Files\Foxmail ...	29M	01:32:30		Foxmail
WmiPrvSE.exe	2852	756		C:\Windows\System32\...	8M	12:33:47		WMI Pro
unsecapp.exe	2908	756		C:\Windows\System32\...	5M	12:33:47		
userAgent.exe *32	2924	2368	SYSTEM	C:\Program Files (x86)\I...	5M	12:33:47		Usernam
conhost.exe	2992	528	SYSTEM	C:\Windows\System32\c...	3M	12:33:47		
webcategory.exe...	3004	2368	SYSTEM	C:\Program Files (x86)\I...	11M	12:33:47		Website
conhost.exe	3048	528	SYSTEM	C:\Windows\System32\c...	3M	12:33:47		
taskhost.exe	3088	632	tanzhe	C:\Windows\System32\t...	17M	12:33:45		

模块 | 线程 | 窗口 | 定时器 | 进程API钩子 | 进程权限

进程ID	挂钩函数	钩子类型	钩子地址	原始模块	钩子模块	原始值	现在值	文件厂商
2848	DefDlgProcA	INLINE HOOK	0x770e1448	user32.dll		4E54444...	FF255010...	
2848	DefDlgProcW	INLINE HOOK	0x770d3b72	user32.dll		4E54444...	FF250811...	
2848	DefWindowProcA	INLINE HOOK	0x770acd12	user32.dll		4E54444...	FF252010...	
2848	DefWindowProcW	INLINE HOOK	0x770a26ad	user32.dll		4E54444...	FF25D810...	
2848	EnableScrollBar	INLINE HOOK	0x748d9cfa	user32.dll		6A10685...	E961805...	
2848	GetScrollInfo	INLINE HOOK	0x748d4e48	user32.dll		6A1068A...	E953CF56...	
2848	GetScrollPos	INLINE HOOK	0x748d5064	user32.dll		8BFF558B...	E977CD5...	
2848	GetScrollRange	INLINE HOOK	0x748da084	user32.dll		8BFF558B...	E9877D5...	

图 20－5　Windows 7 系统中用户态进程的 Inline Hook 实例

77B3D651	90	nop
77B3D652	90	nop
77B3D653	90	nop
77B3D654	90	nop
77B3D655	90	nop
77B3D656	8BFF	mov edi,edi
77B3D658	55	push ebp
77B3D659	8BEC	mov ebp,esp

图 20－6　某应用进程函数的开头部分

在 X86 架构 Windows 系统的编译器中,一般的 API 开头都被编译成“mov edi,edi”“push ebp”和“mov ebp,esp”这样几条指令,其对应的机器码共占用 5 个字节(如图 20－6 所示),这对于一条跳转指令而言是刚刚好的长度。而有时候这几条固定指令之前可能还有一些空操作指令 nop,每条 nop 指令只占一个字节,这些都是可以利用的进行 Inline Hook 的点。其实无论是开头的空操作还是 nop 指令后面的几条汇编指令,都是

Windows 编译器的有意为之,以方便对函数开头进行挂钩。

不过随着操作系统的发展,在内核层进行 Inline Hook 越来越困难重重。高版本的 Windows 乃至其他操作系统都对可执行内存加入了诸多监视,例如数据执行保护(Data Execution Prevention,DEP)策略能够在内存上执行额外检查以防止在不可执行的内存页面上运行恶意代码,而我们进行 Inline Hook 的许多代码在 Windows 眼中都是恶意的、有悖于操作系统安全而运行的代码。检测 Inline Hook 的手段也异常简单粗暴:只需要对比内存映像文件和原始磁盘文件的异同就可以了。

Detours 是微软开发的一个函数库,是专门用来进行 Inline Hook 的,其挂钩过程如图 20-7 所示。对于一个函数,Detours 一般使用 jmp 或 call 指令来覆盖函数的开头部分,并跳转到用户指定的代码区域。被替换的函数开头的几个字节被保存到 Trampoline(跳板)函数中,也就是说 Trampoline 保存了被替换的函数的前几条指令和一个跳转指令,以便于跳转到目标函数的剩余指令中。当执行完 Inline Hook 指定的代码后既可以直接返回,也可以调用 Trampoline,让 Trampoline 调用原函数。可以说 Detours 为 Inline Hook 提供了很大的方便,并且可以免于 DEP 等机制的筛查。



图 20-7 Detours 挂钩过程

20.1.2 SSDT Hook

SSDT 就是系统服务描述符表,这是 X86/X64 体系下固有的数据结构。Windows 系统中有两个 SSDT,一个是 ntoskrnl. exe 导出的 ServiceDescriptorTable,另一个是在 win32k. sys 中存在但并不导出的 ServiceDescriptorTableShadow,这在前面的章节中已经有所描述。SSDT Hook 的挂钩方式与 Inline Hook 不同,它是通过替换表中的内核函数入口地址的方式实现挂钩的。

从图 20-8 可以看出,在本机的 Windows 7 环境下,SSDT 的挂钩数量为 0。在当前的技术条件下,SSDT Hook 作为一种比较过时的挂钩手段,无论是在进攻方一侧还是在防御方一侧,都不再被大规模使用了。首先 SSDT 作为一种重要的内核资源,对其挂钩必然会带来效率上的损失,同时也带来了稳定性问题,SSDT 的任何风吹草动都会造成 BSOD 这样的严重后果;再者 SSDT 虽为内核资源,但却处于内核中最接近用户态进程的位置,而有些攻击方将 Rootkit 植入得很深,处于更为底层的驱动、I/O 管理器等层次,因此在 SSDT 中挂钩以检测异

常行为也起不到很好的作用。

序号	函数名称	当前函数地址	是否被挂钩	当前函数所在的模块	文件厂商
0	NtMapUserPhysicalPagesScatter	0xfffff80004aa7820		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
1	NtWaitForSingleObject	0xfffff80004938ad0		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
2	NtCallbackReturn	0xfffff800046e6f40		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
3	NtReadFile	0xfffff8000493c200		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
4	NtDeviceIoControlFile	0xfffff8000499ac80		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
5	NtWriteFile	0xfffff8000493cc10		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
6	NtRemoveIoCompletion	0xfffff8000493bc40		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
7	NtReleaseSemaphore	0xfffff80004954f50		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
8	NtReplyWaitReceivePort	0xfffff8000492d060		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
9	NtReplyPort	0xfffff80004a7e300		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
10	NtSetInformationThread	0xfffff80004935850		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
11	NtSetEvent	0xfffff8000494b6ec		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
12	NtClose	0xfffff80004938010		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
13	NtQueryObject	0xfffff80004950058		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
14	NtQueryInformationFile	0xfffff80004b075a0		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
15	NtOpenKey	0xfffff8000492a898		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
16	NtEnumerateValueKey	0xfffff80004930790		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
17	NtFindAtom	0xfffff80004999660		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
18	NtQueryDefaultLocale	0xfffff800049162b4		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
19	NtQueryKey	0xfffff80004934c20		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
20	NtQueryValueKey	0xfffff80004936f00		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
21	NtAllocateVirtualMemory	0xfffff80004aea1b0		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...
22	NtQueryInformationProcess	0xfffff80004b1f940		C:\Windows\system32\ntoskrnl.exe	Microsoft Co...

函数个数：401，被挂函数个数：0

图 20－8　Windows 7 系统中的 SSDT

这里要注意的是 SSDT 不是一个表，而是包含了多个表指针的数据结构，其各个成员变量如下所示：

```
typedef struct_ServiceDescriptorTable {
  PVOID ServiceTableBase;                //系统服务分发表的基地址
  PVOID ServiceCounterTable;             //包含着 SSDT 中每个服务被调用次数的计数器，这个计数
                                         //器一般由 sysenter 更新
  unsigned int NumberOfServices;         //由 ServiceTableBase 描述的服务的数目
  PVOID ParamTableBase;                  //包含每个系统服务参数字节数表的基地址——系统服务参
                                         //数表
} * PServiceDescriptorTable;
```

SSDT 中只有第一个元素 ServiceTableBase 和第三个元素 NumberOfServices 才是数组指针，图 20－8 中所列举的各个系统调用都是存放在由 ServiceTableBase 指针指向的函数数组中的。为了行文方便在本章中我们笼统地称这个数组为 SSDT。

对 SSDT 进行 Hook 同样也要突破内存的写保护限制。因为 SSDT 区域属于内核可执行区域，存在与 Inline Hook 相同的问题。这里的解决思路也与前文描述的一致，即先修改内存写保护属性为可写，更改 SSDT 中需要挂钩的函数指针，再恢复之前的内存写保护级别。这里有几个问题需要注意：

➢ 如何修改内存写保护？

➢ 如何找到 SSDT？

➢ 修改成什么样的函数？

这三个问题也是 SSDT Hook 的实际工程问题，这里的解决思路如下：

（1）针对如何修改内存写保护，可以采用内存描述符表（MDL），MDL 用于描述连续的

虚拟内存,这正好可以用于 SSDT 的修改:

① 采用 IoAllocateMdl 方法分配一个 MDL 区域,用以将 KeServiceDescriptorTable 映射到一块虚拟内存中。

② 采用 MmBuildMdlForNonPagedPool 方法构造和初始化 MDL 页帧号数组,这个数组是在非分页内存池中构造的。

③ 经过上述两步后,SSDT 所占的物理内存就在我们的内存视图中映射好了,此时在这个视图中直接修改函数指针就可以了。

④ 修改完成后要使用 MmUnlockPages 和 IoFreeMdl 方法释放 MDL。

(2) 针对如何找到 SSDT,又可以采用三种方式实现:

① 利用 KPCB(内核进程控制块)找到 SSDT:

- 在 X86/X64 系统中,FS[0]寄存器指向处理器控制区(KPCR),这个控制区保存了系统中每个进程的信息。
- KPCR 中有个成员变量 PrcbData 指向处理器控制块 KPRCB,KPRCB 也是放在 KPCR 后面的数据结构。
- KPRCB 包含了当前线程的 KTHREAD 结构,KTHREAD 中的 ServiceTable 指针指向 SSDT(若为 GUI 线程则指向 ShadowSSDT)。
- 总结一下,其查找顺序为 FS[0]→KPCR→KPRCB→KTHREAD→SSDT。

② 使用 ETHREAD 查找 SSDT:ETHREAD 包含了指向 KTHREAD 的指针,其查找顺序可以为 ETHREAD→KTHREAD→SSDT,这是一种比较简便的方式。

③ 引用 ntoskrnl. exe 的输出变量动态获取 SSDT(但无法获取 ShadowSSDT)。

(3) 针对修改成什么样的函数,这个答案是比较固定的,即参数数量和类型要一致,调用约定要一致。在 SSDT 中的系统调用的调用约定都是 stdcall 类型的。

对于 SSDT Hook,可以采用遍历的方式进行检测。在 SSDT 中的系统调用的地址范围都不能超过 ntoskrnl. exe 模块的地址范围,否则就可以认为被挂钩了。如图 20 - 9 所示,在 64 位 Windows 7 系统下,ntoskrnl. exe 的基址和大小都有了,模块的地址范围也就有了。

驱动名	驱动类型	基地址	大小	驱动对象	驱动路径	文件厂商
ntoskrnl.exe	过滤驱动	0xfffff80004650000	0x5dd000	-	C:\Windows\system32\n...	Microsoft Coi
Ntfs.sys	过滤驱动	0xfffff8800124a000	0x1a7000	0xfffffa8007248300	C:\Windows\system32\D...	Microsoft Coi

图 20 - 9　ntoskrnl. exe 的基址和大小

20.1.3　IDT Hook

IDT(Interrupt Descriptor Table,中断描述符表)是 Windows 系统中的重要数据结构,所有的中断服务例程都存放于 IDT 中。与 SSDT 一样,IDT 也位于 ntoskrnl. exe 模块,挂钩 IDT 可以截断中断处理流程,这在网络数据包深度检测、中断信号处理等领域很有意义。在 X86 系统中 IDT 是由 8 字节长的描述符组成的一个数组,数组中的每个元素被称为 IDT_ENTRY,其结构如图 20 - 10 所示,而 64 位 Windows 7 系统下的 IDT 则如图 20 - 11 所示。

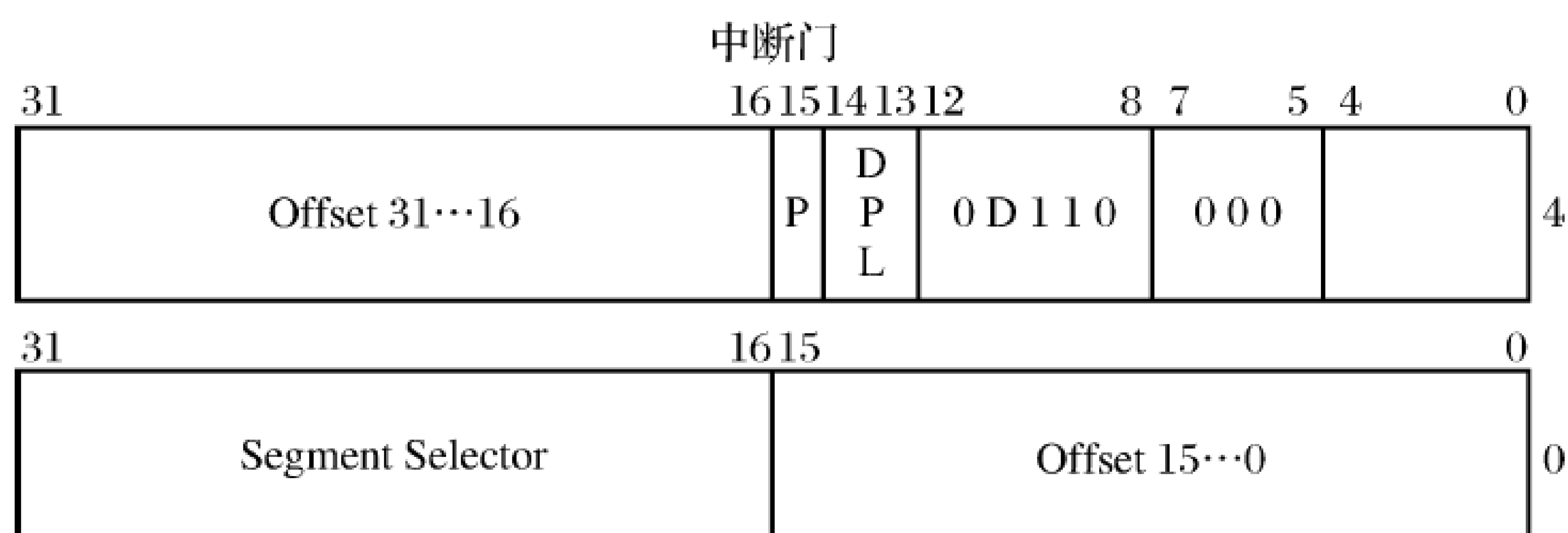


图 20－10　IDT_ENTRY 结构

每个 IDT_ENTRY 的中断处理例程的地址偏移(Offset)被分成了高位部分和低位部分，相应地 IDT Hook 要对每一项的 IDT_ENTRY 同时挂钩高位地址和低位地址两部分。

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

Cpu...	序号	函数名称	段选择子	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
0	00	Divide error	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	01	Debug	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	02	Not used	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	03	Breakpoint	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	04	Overflow	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	05	Bounds check	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	06	Invalid opcode	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	07	Device not available	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	08	Double fault	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	09	Coprocessor segment over...	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	0A	Invalid TSS	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	0B	Segment not present	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	0C	Stack segment fault	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	0D	General protection	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	0E	Page Fault	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	0F	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	10	Floating-point error	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	11	Alignment check	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	12	Machine check	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	13	SIMD floating point exception	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	14	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	15	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	16	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	17	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	18	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	19	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	1A	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	1B	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	1C	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	1D	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe
0	1E	Reserved by Intel	0x02	0xFFFFF80004821...	-	0xFFFFF80004821...	C:\Windows\system32\ntoskrnl.exe

IDT派发函数：1024，被挂钩函数：0

图 20－11　Windows 7 系统中的 IDT

改写 IDT 的方式与 SSDT Hook 一样，都是要先解决内存可写性的问题。不过这里采取了一种不同的方式，即使用 CR0 寄存器的写保护位实现内存可写性。

CR0 寄存器是控制寄存器，其中第 16 位是写保护位 WP，只要将其置 0 就可以禁用写保护，置 1 则可将其恢复。具体代码如下：

```
_asm{
    mov eax,cr0
    push eax
    and eax,0xfffeffff      //将第16位(写保护位)置为0
    mov cr0,eax
}
```

恢复原有保护级别的代码如下：

```
_asm{
    pop eax
```

```
        mov cr0,eax
    }
```

至于怎么覆写,其流程就与 SSDT Hook 如出一辙了。

对于 IDT Hook 的检测,主要是查看存储在 IDT 中的 0x2E 项(中断方式系统调用的总入口)的地址是否处于内核模块 ntoskrnl. exe 或 ntkrnlpa. exe 的范围内,也要检测某些硬件设备中断向量号对应的函数入口地址是否处于合法的驱动模块内:若是,则表明一切正常;若否,则表明 IDT Hook 存在。目前 Windows 操作系统已经废弃了中断方式的系统调用,取而代之的是快速调用,但硬件中断的 ISR 还是保留在 IDT 中的,用这种方式进行挂钩也是内核挂钩比较常用的做法。

20.1.4　GDT Hook

GDT(Global Descriptor Table,全局描述符表)是 Windows 中的重要数据结构。GDT 与 LDT(Local Descriptor Table,本地描述符表)共同构成了段选择子的描述体系,可用于描述远跳转和跨段的场景:根据 CS 寄存器中段选择子的不同,选择 GDT/LDT 中不同的代码段或数据段。GDT 全局只有一份,LDT 却可以有多份,Windows 7 系统下的 GDT 如图 20-12 所示。

Cpu...	段选...	基址	界限	段...	段...	类型
0	0x0002	0x0000000000...	0xFFFFFFFFFFF...	Byte	0	Code RE Ac
0	0x0003	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Data RW Ac
0	0x0004	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Code RE Ac
0	0x0005	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Data RW Ac
0	0x0006	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Code RE Ac
0	0x0008	0x00F8800004...	0xFFFFFFFFFFF...	Byte	0	T5532 Busy
0	0x000A	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Data RW Ac
0	0x000C	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Code RE
1	0x0002	0x0000000000...	0xFFFFFFFFFFF...	Byte	0	Code RE Ac
1	0x0003	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Data RW Ac
1	0x0004	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Code RE Ac
1	0x0005	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Data RW Ac
1	0x0006	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Code RE Ac
1	0x0008	0x00F8800004...	0xFFFFFFFFFFF...	Byte	0	T5532 Busy
1	0x000A	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Data RW Ac
1	0x000C	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Code RE
2	0x0002	0x0000000000...	0xFFFFFFFFFFF...	Byte	0	Code RE Ac
2	0x0003	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Data RW Ac
2	0x0004	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Code RE Ac
2	0x0005	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Data RW Ac
2	0x0006	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Code RE Ac
2	0x0008	0x00F8800004...	0xFFFFFFFFFFF...	Byte	0	T5532 Busy
2	0x000A	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Data RW Ac
2	0x000C	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Code RE
3	0x0002	0x0000000000...	0xFFFFFFFFFFF...	Byte	0	Code RE Ac
3	0x0003	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Data RW Ac
3	0x0004	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Code RE Ac
3	0x0005	0x0000000000...	0xFFFFFFFFFFF...	Page	3	Data RW Ac
3	0x0006	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Code RE Ac
3	0x0008	0x00F8800004...	0xFFFFFFFFFFF...	Byte	0	T5532 Busy
3	0x000A	0x0000000000...	0xFFFFFFFFFFF...	Byte	3	Data RW Ac
3	0x000C	0x0000000000...	0xFFFFFFFFFFF...	Page	0	Code RE

GDT段数量：32，可疑项数：0.

图 20-12　Windows 7 系统中的 GDT

在作者所用 Windows 7 环境下 GDT 中共有 32 个有效项，每一项称为一个全局描述符，全局描述符又可细分为数据段描述符（如图 20－13 所示）、程序段描述符、系统段描述符、门描述符（如图 20－14 所示）等种类，其基址（Base）也是被分成了高址和地址两部分，但根据描述符种类的不同高址和低址的划分也不尽相同。

之所以要采用门描述符的方式描述段间跳转，主要是因为在跳转的过程中每个段都要有相应的校验和特权级检查工作，而段描述子中除了记载跳转地址，也记录了特权级检查和校验的信息。

本质上 GDT Hook 就是在 GDT 中插入新的调用门描述符来获取执行特权，但一般不会改写 GDT 中已有的描述符。因为 GDT 太重要了，以至于不允许修改有效表项而只允许扩展表项，否则可能造成 GDT 不稳定的问题，甚至在修改的过程中造成 BSOD。可以在有效表项的后面新增新的门描述符进行扩展。

基于此，GDT Hook 的步骤是：通过 GDTR 获取 GDT→获取 GDT 中的空置项→保存原始项以便后面进行恢复→将新增的门描述符覆写到该空置项的位置上。

图 20－13　GDT 中的数据段描述符

图 20－14　GDT 中的调用门描述符

20.1.5　MSR Hook

MSR（Model Specific Register，专有寄存器）是在 Pentium II CPU 之后引入的特殊寄存器，用于改善系统调用的性能，而在此之前的系统调用是采用 int 0x2E 中断指令进行的，并且需要使用堆栈保存不可变寄存器的内容。

MSR 寄存器共有 3 个，如表 20－1 所示。

表 20 − 1　MSR 寄存器

寄存器	功能	代号
SYSENTER_CS_MSR	保存跳转之后的 cs 内容	174h
SYSENTER_CS_ESR	保存跳转之后的 eps 内容	175h
SYSENTER_CS_EIP	保存目标地址	176h

Pentium II CPU 也引入了对应的新的系统调用/调用退出指令：sysenter 和 sysexit。同时，系统还支持扩展指令 rdmsr 和 wrmsr 以用于 MSR 的读写。采用 rdmsr 读时会将 MSR 中的 64 位的信息读取到（EDX：EAX）寄存器中，EDX 存放高 32 位部分，EAX 存放低 32 位部分；采用 wrmsr 写时可将要写的信息存入（EDX：EAX）寄存器中，同样也是 EDX 存放高 32 位部分，EAX 存放低 32 位部分（在 64 位系统下，使用 RDX 和 RAX 代替 EDX 和 EAX 寄存器，RDX 和 RAX 的高 32 位置零，只使用低 32 位存储 MSR 的数据）。在读写时 ECX 寄存器要填写目标寄存器代号，例如 0x174 代表了 SYSENTER_CS_MSR 寄存器。有了保存的地址和目标寄存器，读写操作才能正常进行。

MSR Hook 也被称为 KiFastCallEntry-Hook，其主要思想是干涉 EIP 寄存器，使系统调用时的指令走向发生改变，因此 MSR Hook 的主要目标是 0x176 号寄存器，即 SYSENTER_CS_EIP，这是快速调用方式中的指令指针寄存器，其作用是保存要跳转的指令指针。每当执行 sysenter 指令时，sysenter 都会将 SYSENTER_CS_EIP 寄存器中的内容填写到 EIP 寄存器中。因此改变了这个寄存器的内容也就改变了 EIP 寄存器的源内容，也就可以改变系统调用的执行流程。

挂钩过程中先使用 rdmsr 指令读取 SYSENTER_CS_EIP 寄存器的内容并保存起来，之后使用 wrmsr 指令将挂钩的指令指针写到 SYSENTER_CS_EIP 中，只需要两步就完成了 MSR 的 Hook。当然在 Hook 之前要压栈保存 EDX、EAX、ECX 这几个可变寄存器的内容。由于 MSR Hook 是内核态挂钩，MSR Hook 只能在 Ring0 或实模式下进行，因此应使用内核态驱动的方式实现。

检测 MSR Hook 的方式也非常简单，就是要检查 SYSENTER_CS_EIP 寄存器中的内容是否在 ntoskrnl. exe 或 ntkrnlpa. exe 范围内。因为 SYSENTER_CS_EIP 寄存器执行的是系统调用，而系统调用都是在内核模块中，也就是上述两个模块之中，因此无论怎么跳转都不应该“出圈”。

20.1.6　IAT/EAT Hook

IAT 就是导入地址表。在 PE 文件中会有各种依赖关系，例如 DLL1 依赖于 DLL2，这是非常常见的，所谓依赖，最直接的体现就是依赖一些函数接口。IAT 就是描述所依赖的函数接口地址的表。

EAT 就是导出地址表。PE 文件有时也会导出一些函数供其他模块调用，例如图 20 − 15 中的导入地址表中的函数，就是由本地系统中的 AnalyzeData. dll 模块导出的，如图 20 − 16

所示。一个模块引用另一个模块中的函数，对于前者来说叫作导入，对于后者来说叫作导出。一般来说导入的函数是对应模块导出的函数的子集。

SIPGateModule.dll

Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
0033CD40	N/A	00335EF8	00335EFC	00335F00	00335F04	00335F08
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
AnalyzeData.dll	4	003362CC	00000000	00000000	0033CD40	002E235C

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
0033CCFA	0033CCFA	0006	AnalyzeDataInputData
0033CD12	0033CD12	0009	AnalyzeDataOpenStreamEx
0033CD2C	0033CD2C	0000	AnalyzeDataClose
0033CCE0	0033CCE0	0003	AnalyzeDataGetPacketEx

图 20－15　用户态某模块的导入地址表

AnalyzeData.dll

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00001055	0000	00015876	AnalyzeDataClose
00000002	0000108C	0001	00015887	AnalyzeDataGetLastError
00000003	0000106E	0002	0001589F	AnalyzeDataGetPacket
00000004	000011A4	0003	000158B4	AnalyzeDataGetPacketEx
00000005	00001028	0004	000158CB	AnalyzeDataGetSafeHandle
00000006	000011C7	0005	000158E4	AnalyzeDataGetTail
00000007	00001050	0006	000158F7	AnalyzeDataInputData
00000008	0000110E	0007	0001590C	AnalyzeDataOpenFile

图 20－16　用户态某模块的导出地址表

IAT/EAT 的具体格式和形态在第 12 章已有详细描述。IAT Hook 就是将 IAT 中欲导入模块的函数入口地址替换为其他模块的入口地址。IAT Hook 是比较常见的挂钩方式，也是一种常见于用户态挂钩的方式。

在被导入模块加载到进程内存空间之前，IAT 与 INT 都指向相同的内容；而当被导入模块加载到进程内存空间之后，两者分别指向了不同的内容：IAT 指向导入模块的函数入口地址，INT 仍然指向函数名，且在进程运行中使用 IAT 进行寻址。而我们要挂钩的就是这个 IAT 中的地址，这是在内存中以“热”的方式替换的。EAT 的挂钩方式与 IAT 的大致一样。

因此，挂钩 IAT/EAT 的步骤是：在内存中定位 IAT/EAT→保存表中原始的导入/导出函数地址项→用新的函数地址替换相应地址项→完成挂钩业务后恢复原始的导入/导出函数地址项。而对于修改内存的读写权限属性，完全可以通过 VirtualProtect 方法进行更改。

挂钩 IAT/EAT 也面临着与挂钩 SSDT 同样的问题，即参数数量和类型一致性、函数调用约定一致性等问题。

对于 IAT Hook 的检测主要是确定 IAT 中函数的地址是否在被引用的模块的地址范围内。IAT 对于每个被导入的模块都会有一个导入函数表，每一个导入函数表中的地址都应该在对应模块的内存地址范围内，否则就存在 IAT Hook。而对于 EAT Hook，就是要检测导出地址表中的函数是否都在函数宿主模块的地址范围内，不在就说明存在 EAT Hook。

20.1.7　消息报文 Hook

Windows 是采用消息机制驱动的，因此消息是 Windows 的源动力。所谓消息报文 Hook 就是对各种消息的处理进行挂钩，一般是通过 SetWindowsHookEx 这个方法实现的。

SetWindowsHookEx 是 Windows 提供的一个特定 API，由 user32. dll 导出（如图 20 – 17 所示），其核心操作是系统调用 NtUserSetWindowsHookEx，目的在于设置回调例程以监控指定窗口的某种消息。当消息到达后，可以先于目标窗口处理函数去处理到达的消息。消息报文 Hook 机制允许应用程序截获处理 Window（窗口）消息或特定事件。

File: user32.dll
Dos 头部
Nt 头部
文件头部
可选头部
数据目录 [x]
段头部 [x]
输出目录
导入目录
资源目录
异常目录
重定位目录

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	00082938	00084014	000838B0	0008711F
(nFunctions)	Dword	Word	Dword	szAnsi
000008BF	00028BD0	02E3	0008870D	SetWindowsHookExA
000008C0	0000F804	02E4	0008871F	SetWindowsHookExW
000008C1	00040880	02E5	00088731	SetWindowsHookW
000008C2	0002B320	02E6	00088741	SfmDxBindSwapChain
000008C3	0002B330	02E7	00088754	SfmDxGetSwapChainStats

图 20 – 17　user32. dll 导出 SetWindowsHookEx

SetWindowsHookEx 方法将一个由应用程序定义的挂钩安装到挂钩链中去，这个挂钩本质上是由用户定义的回调例程，可以通过这种通知式回调例程对系统中选定的消息或事件进行监控和截获，这些事件既可以与某个特定的线程有关，也可以是系统中的全局事件。

函数的原型是 SetWindowsHookEx（int idHook, HOOKPROC lpfn, HINSTANCE hMod, DWORD dwThreadId），下面分别说明其参数。

➢ **idHook**：表示需要安装挂钩的消息报文的类型，如图 20 – 18 所示，其具体实参枚举如下：

- WH_CALLWNDPROC：用于钩住发送到视窗进程的消息报文，支持在视窗例程处理该报文之前挂钩。
- WH_CALLWNDPROCRET：用于钩住已经发送到视窗进程的消息报文，即在视窗例程处理之后挂钩。
- WH_CBT：用于截获 CBT 事件之前的消息，CBT 事件包括激活、建立、销毁、最小化、最大化、移动、改变尺寸等窗口事件，还包括完成系统指令、移动鼠标、设置鼠标焦点、键盘等事件。
- WH_KEYBOARD：对键盘击键消息进行截获。
- WH_KEYBOARD_LL：对底层的键盘输入事件进行截获，但只能在 Windows NT 中安装。

- WH_MOUSE:支持对鼠标消息进行截获。
- WH_MOUSE_LL:支持对底层的鼠标输入事件进行截获,但只能在 Windows NT 中安装。
- WH_MSGFILTER:支持截获由对话框、消息框、菜单栏或滚动条中的输入事件引发的消息。
- WH_FOREGROUNDIDLE:支持当应用程序的前台线程即将进入空闲状态时截获相关事件和信息。
- 除此之外,还支持 WH_DEBUG、WH_GETMESSAGE、WH_JOURNALRECORD、WH_JOURNALPLAYBACKWH_SHELL、WH_SYSMSGFILTER 等事件报文。

➢ **lpfn**:指向相应的挂钩处理函数指针,在这个处理函数中可以加载 DLL 模块或进行其他操作。

➢ **hMod**:指向一个动态链接库的句柄,该动态链接库包含了参数 lpfn 所指向的挂钩处理例程。

➢ **dwThreadId**:指示了一个线程标识符,该线程与挂钩处理例程相关。

消息钩子 | 进程钩子 | 内核回调表

句柄	类型	函数	模块名	线程Id	进程Id	进程路径
0x0001007A	WH_KEYBOARD_LL	0x000007FE...	hotkey.dll	3188	3184	C:\Windows\System32\rundll32.exe
0x00040181	WH_KEYBOARD_LL	0x000007FE...	tpnumlkd.dll	3536	3532	C:\PROGRA~1\Lenovo\HOTKEY\tpnumlkd.exe
0x03C705C9	WH_KEYBOARD_LL	0x00000000...	WeChatWin.dll	3644	3908	C:\Program Files (x86)\Tencent\WeChat\WeChat.exe
0x03B70CE1	WH_KEYBOARD_LL	0x00000000...	Maxthon.dll	7248	10744	C:\Program Files (x86)\Maxthon\Bin\Maxthon.exe

图 20－18　某 Windows 7 系统下被挂钩的消息及其挂钩宿主进程

每一个挂钩都有一个与之相关联的数据结构列表,称之为钩子链表,这个链表由操作系统来维护。链表中的元素都是应用程序定义的、被挂钩处理例程调用的回调函数,也就是该消息钩子的各个处理例程。因为每一种消息可能不仅仅只有一个钩子,当有多个钩子的时候就需要一个链表来存储。最后安装的钩子存放在链表的起始位置,而最早安装的钩子则放在最后,因此后加入的钩子先获得消息控制权。当有指定类型的消息发生时,系统就把这个消息传递到各个钩子处理例程。

挂钩处理例程可以只监视消息而不做修改,也可以修改消息或者阻断消息的下发,以避免这些消息传递到下一个挂钩处理例程或者目标窗口。如果在回调函数中希望将事件传递到链表中的下一个钩子,可以调用 CallNextHookEx 函数进行下发,而 UnHookWindowsHookEx 则是卸载钩子所使用的方法。

Windows 并不要求挂钩处理例程的卸载顺序一定得和安装顺序相同或相反。每当有一个钩子被卸载,Windows 便释放其占用的内存并更新整个钩子链表。如果进程安装了某个钩子却在尚未卸载钩子之前就终止了,那么系统会自动为该进程卸载钩子。

钩子处理例程是一个应用程序定义的回调函数,但不能定义成某个类的成员函数,而只能定义为全局的 C 风格的函数,用以监控某一特定类型的消息。这些消息可以与某一特定线程相关联,也可以是系统中所有线程的消息,也就是说挂钩既支持局部的消息也支持全局

的消息。

20.1.8　Object Hook

Object Hook 就是对象挂钩,但挂钩的并不是对象本身,而是对象的解析函数。在 Windows 系统中,OBJECT_TYPE_INITIALIZER 是个很重要的数据结构,其中包含了每种对象的处理函数指针。无论是对象的新建还是打开,抑或关闭,都要执行这个数据结构中的成员函数,OBJECT_TYPE_INITIALIZER 结构如下所示:

```
typedef struct_OBJECT_TYPE_INITIALIZER {
  USHORT Length;
  BOOLEAN UseDefaultObject;
  BOOLEAN CaseInsensitive;
  ULONG InvalidAttributes;
  GENERIC_MAPPING GenericMapping;
  ULONG ValidAccessMask;
  BOOLEAN SecurityRequired;
  BOOLEAN MaintainHandleCount;
  BOOLEAN MaintainTypeList;
  POOL_TYPE PoolType;
  ULONG DefaultPagedPoolCharge;
  ULONG DefaultNonPagedPoolCharge;
  PVOID DumpProcedure;          //函数指针,指向对象 dump 处理过程
  PVOID OpenProcedure;          //函数指针,指向对象打开处理过程
  PVOID CloseProcedure;         //函数指针,指向对象关闭处理过程
  PVOID DeleteProcedure;        //函数指针,指向对象删除处理过程
  PVOID ParseProcedure;         //函数指针,指向对象解析处理过程
  PVOID SecurityProcedure;
  PVOID QueryNameProcedure;
  PVOID OkayToCloseProcedure;
} OBJECT_TYPE_INITIALIZER, * POBJECT_TYPE_INITIALIZER;
```

OBJECT_TYPE_INITIALIZER 包含在 OBJECT_TYPE(对象类型)结构中,而 OBJECT_TYPE 又是 OBJECT_HEADER(对象头)结构的成员变量,也就是说,任何 Windows 中的对象都会包含 OBJECT_TYPE_INITIALIZER 这个数据结构。这是因为系统中每个对象都有对象头结构 OBJECT_HEADER,不管是什么类型的对象,不同的只是对象体,对象头总是相同的。

打开/关闭文件时要遵循以下流程:

(1) 执行系统调用 NtCreateFile。

(2) 调用 I/O 管理器中的 IoCreateFile。

(3) 调用对象管理器的 ObOpenObjectByName。

(4) 调用对象管理器的对象查找函数 ObpLookupObjectName。

(5) 执行文件解析函数 IopParseFile。

(6) 执行设备解析函数 IopParseDevice。

最后一步的 IopParseDevice 最终要找到目标类型的对象执行 OBJECT_TYPE_INITIALIZER 中的 OpenProcedure 函数。这里包含了两层含义:

- 对于对象的打开/关闭操作,最终还是要调用具体对象头中定义的打开/关闭处理例程,因为只有每种对象才最懂自己,知道怎样打开和关闭自己。
- 不同种类的对象可能会有不同的对象处理例程。

而我们所说的 Object Hook，就是挂钩上述 OBJECT_TYPE_INITIALIZER 中的处理例程（OpenProcedure、CloseProcedure 等），在这些例程中“做手脚”，可以有效截获对象处理的相关操作。

图 20－19 中展示了一个 Object Hook 实例。

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

函数名	当前函数地址	Hook	原始函...	Object类型	Object地址	当前函数地址所在模块
SeDefaultObjectMethod	0xFFFFF80004...	-	-	FilterCommunicatio...	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
CloseProcedure	0xFFFFF80004...	-	-	PowerRequest	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
SeDefaultObjectMethod	0xFFFFF80004...	-	-	PowerRequest	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
OpenProcedure	0xFFFFF80004...	-	-	TmRm	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
CloseProcedure	0xFFFFF80004...	-	-	TmRm	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
DeleteProcedure	0xFFFFF80004...	-	-	TmRm	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
SeDefaultObjectMethod	0xFFFFF80004...	-	-	TmRm	0xFFFFFA8006...	C:\Windows\system32\ntoskrnl.exe
PostOperation	0xFFFFF88004...	ObjectType_Callback	-	Process	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360FsFlt.sys
PreOperation	0xFFFFF88004...	ObjectType_Callback	-	Process	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360FsFlt.sys
PreOperation	0xFFFFF88008...	ObjectType_Callback	-	Process	0xFFFFFA8006...	C:\Windows\system32\drivers\QQProtectX64...
PostOperation	0xFFFFF88004...	ObjectType_Callback	-	Process	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360Box64.sys
PreOperation	0xFFFFF88004...	ObjectType_Callback	-	Process	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360Box64.sys
PostOperation	0xFFFFF88004...	ObjectType_Callback	-	Thread	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360FsFlt.sys
PreOperation	0xFFFFF88004...	ObjectType_Callback	-	Thread	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360FsFlt.sys
PreOperation	0xFFFFF88008...	ObjectType_Callback	-	Thread	0xFFFFFA8006...	C:\Windows\system32\drivers\QQProtectX64...
PostOperation	0xFFFFF88004...	ObjectType_Callback	-	Thread	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360Box64.sys
PreOperation	0xFFFFF88004...	ObjectType_Callback	-	Thread	0xFFFFFA8006...	C:\Windows\system32\DRIVERS\360Box64.sys
GetCellRoutine	0xFFFFF80004...	-	0xFFFF...	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpAllocate	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFree	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileSetSize	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileWrite	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileRead	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileFlush	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
GetCellRoutine	0xFFFFF80004...	-	0xFFFF...	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpAllocate	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFree	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileSetSize	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileWrite	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileRead	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe
CmpFileFlush	0xFFFFF80004...	-	-	HHIVE	0xFFFFF8A000...	C:\Windows\system32\ntoskrnl.exe

Object Type函数： 261，被挂钩函数： 15

图 20－19　某 Windows 7 系统下的 Object Hook 实例

ObRegisterCallbacks 是对象管理器中的函数，用于注册各种对象的回调函数，其第一个参数为 POB_CALLBACK_REGISTRATION 类型，也就是 OB_CALLBACK_REGISTRATION 数据结构的指针，而 OB_CALLBACK_REGISTRATION 结构又包含了 OB_OPERATION_REGISTRATION 结构体，后者是一个回调函数信息结构体，其布局如下所示：

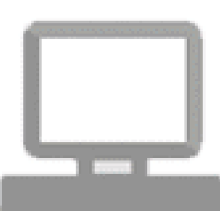
```
typedef struct_OB_OPERATION_REGISTRATION {
    __in POBJECT_TYPE    *ObjectType;
//对象的类型，这里注册回调函数的类型 PsProcessType 和 PsThreadType（进程对象和线程对象）
    __in OB_OPERATION  Operations;
//注册回调的操作方式，一个是创建进程，一个是拷贝进程句柄
    __in POB_PRE_OPERATION_CALLBACK  PreOperation;
//创建之前回调函数的地址，每一个回调包含什么信息在本结构体中给出
    __in POB_POST_OPERATION_CALLBACK  PostOperation;
//创建之后回调函数的地址
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

也就是说，ObRegisterCallbacks 用于注册各种对象的回调函数，最核心的操作就是注册 OB_OPERATION_REGISTRATION 数据结构，这个数据结构里包含了回调函数的各种信息。

例如当向目标进程添加保护权限时，可以在安装挂钩时通过 ObRegisterCallbacks 方法设置 OB_OPERATION_REGISTRATION 的前序操作和后序操作函数（PreOperation 和 PostOperation）这两个成员变量，以拦截对象创建前和对象创建后的通知信息。

从图 20－19 中可以看出，此处挂钩的目标是进程和线程对象，挂钩目标函数是前序和





后序操作例程 PreOperation 和 PostOperation，而挂钩后的函数名为“ObjectType_Callback”，挂钩函数所在模块的所属进程为 360 和 QQ 的系统保护软件驱动程序，因此挂钩操作的实施进程应为 360 和 QQ 的系统保护进程，并由此可以推断这是 360 和 QQ 的系统保护软件对于自身的打开和关闭的监控与保护。

在 Windows Vista 之后，由于 PatchGuard 机制的大规模应用，通过 ObRegisterCallbacks 和 ObUnRegisterCallbacks 函数注册/反注册回调钩子已经成了主流选择。

20.1.9　DKOM

DKOM（Direct Kernel Object Manipulation，直接内核对象操纵）是另一类 Rootkit 手段，其核心思想是直接修改内存中的内核对象，使之能实现诸如进程/线程隐藏、驱动程序隐藏、特权提升等目标。但实现这些目标也要基于一定的系统基础：

- 进程/线程都是在内存中运行的，DKOM 操纵的是内存中的数据结构/内核对象。
- 进程/线程等在内存中都有对应的数据结构（例如 ETHREAD、KTHREAD、EPROCESS、KPROCESS 等），并且这些数据结构是可以修改其中的成员变量的。
- 系统中存在某些全局列表，用以存放 ETHREAD、EPROCESS 等数据结构，并支持将这些结构串联起来。

例如，进程的隐藏或伪造就可以采用 DKOM 的方式。

操作系统中有一个总链表 PsActiveProcessHead，这个链表中存放着系统中所有进程的 EPROCESS 结构，并且这些 EPROCESS 结构通过自身的 ActiveProcessLinks 结构（含前向指针 Flink 和后向指针 Blink）连接在一起构成双向链表，如图 20 - 20 所示。系统管理工具（如任务管理器 taskmgr. exe 等）或系统调用都是通过这个链表来获取系统中所有进程信息的。

基于这种双向链表来枚举所有进程就造成了一种偏听则暗的“灯下黑”：Rootkit 完全可以通过修改双向链表来隐藏进程或伪造进程，比如对中间某个 EPROCESS“断链”，或者伪造一个 EPROCESS“进链”。

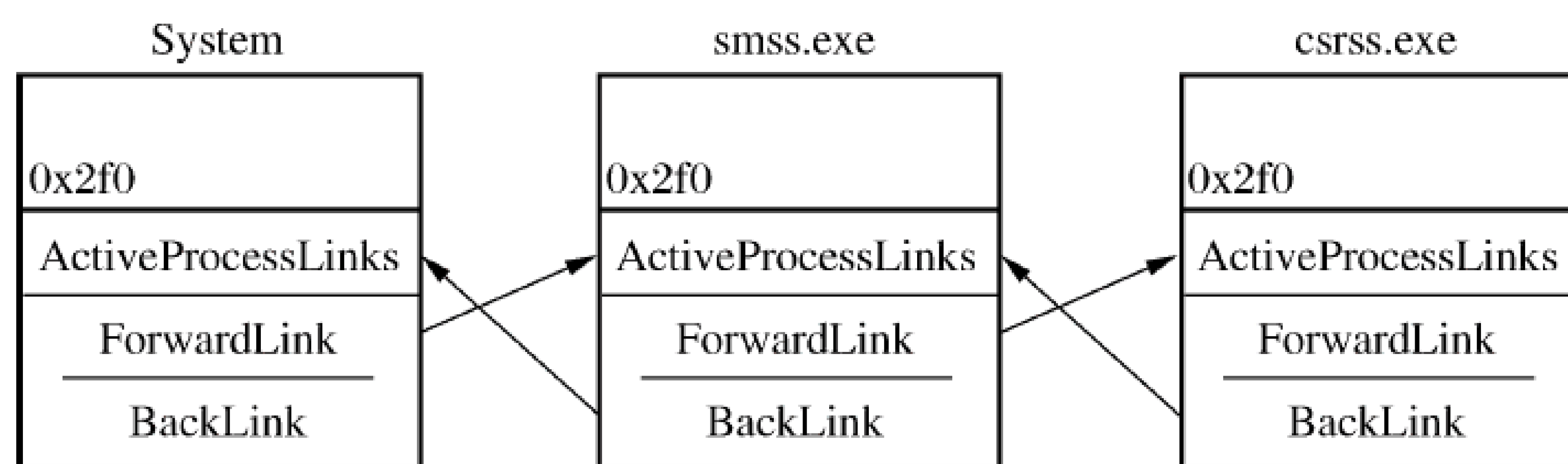


图 20 - 20　系统进程链表 PsActiveProcessHead

基于上述思想，想要隐藏和伪造进程必须首先获取 PsActiveProcessHead 链表，这也是基于 DKOM 技术隐藏进程的关键一步。有以下几种获取方法：

1）从 KdInitSystem 函数地址处硬编码搜索

系统内核变量 KdDebuggerDataBlock 是一个 KDDEBUGGER_DATA64 类型的结构体，其成员变量 PsActiveProcessHead 正指向我们要定位的 PsActiveProcessHead 的地址，因此首要的

任务就是定位 KdDebuggerDataBlock。

由于 KdDebuggerDataBlock 结构被 KdInitSystem 函数引用,KdInitSystem 又被 ntoskrnl. exe 的导出函数 KdEnableDebugger 调用,因此 PsActiveProcessHead 链表最终可通过 ntoskrnl. exe 的导出函数 KdEnableDebugger 获取。这个调用链的流程是:ntoskrnl. exe→KdEnableDebugger→KdInitSystem→KdDebuggerDataBlock→目标地址。

2）从 System 进程(pid =4)的 EPROCESS 地址获取

PsActiveProcessHead 是活动进程链表头,理论上是链表中第二个进程的 EPROCESS 结构成员 ActiveProcessLinks. Blink 指向的目标,同时也是链表中最后一个进程的 EPROCESS 结构成员 ActiveProcessLinks. Flink 指向的目标。

第二个进程即 System 进程(PID =4),因此可以通过 PsLookupProcessByProcessId 从 System 进程推断出目标链表地址。

3）从 ntoskrnl. exe 的导出变量 PsInitialSystemProcess 中获取

ntoskrnl. exe 导出了一个类型为 EPROCESS 指针结构的变量 PsInitialSystemProcess,其指向进程 System(PID =4)的 EPROCESS 结构,因此获取链表的方法与上一个方法类似。

4）从 KPCR 中获取

KPCR 的 0x034 位置的成员 KdVersionBlock 是一个 DBGKD_GET_VERSION64 类型的指针。该结构体成员变量 DebuggerDataList 是 KdpDebuggerDataListHead,KdpDebuggerDataListHead 的 Flink 指针就指向 KdDebuggerDataBlock。这就又回到了第一个方法。

5）调用 NtSystemDebugControl 函数获取

通过 SSDT 中的函数 NtSystemDebugControl 可以获取 DBGKD_GET_VERSION64 数据结构,获取了这个数据结构,查找链表的方法就与上一个方法相同了。

获取链表以后,在链表中操纵内核对象就很简单了,这里就不详细介绍了。

20.1.10 IVT Hook

在 X86/X64 架构下,中断向量(IV)是指实模式下存放在中断向量表中的中断服务例程,而中断向量表(Interrupt Vector Table,IVT)自然也就是实模式下存放中断向量的表格,在此可将 IVT 与 IDT 作一个对比。

IDT 是处于保护模式下的中断服务描述符表,表中的 64 位描述符是在保护模式下才能被识别和使用的,因此 IDT 是操作系统加载完成进入保护模式以后的中断服务例程表。表中每一项都是一个中断描述符,既包括了段选择子(用于从 GDT 或 LDT 中选择段描述符),也包括了段偏移。在保护模式下 IDTR 指向 IDT。

IVT 则是处于实模式下的中断向量表,其中的每一项就是中断服务例程地址而不是中断描述符。因为此时处于实模式下,GDT 或 LDT 尚未初始化,无法通过从 IDT 的中断描述符到 GDT 的段描述符的“二级跳”方式找到中断服务例程,因此表中只能直接存放中断服务例程的入口地址,如图 20 -21 所示。在实模式下也是由 IDTR 指向 IVT 的。

从图 20－21 可以看出，每个中断向量占 4 B。其中中断服务例程入口是以 segment：offset 的形式提供的，offset 在低址端，segment 在高址端，整个 IVT 从地址 0x0 一直延伸到 0x3FF，总共占据 1KB 大小。

IVT Hook 就是替换 IVT 中的某一项，将中断服务例程的入口地址替换为自己定义的函数入口地址；或者干脆整体切换 IVT，从而挂钩实模式下的所有中断服务例程。

例如在实模式下先通过 lidt 指令获取 IDTR 的值以得到 IVT 的基址（base）和界限值（limit），再将该值保存在变量中并设置新的地址变量：界限值不变，IVT 的基址可以设置为自己伪造的 IVT 的基址，最后使用 sidt 指令将上述伪造的 IVT 基址和界限值保存到 IDTR 中。这样就完成了 IVT 的整体切换。

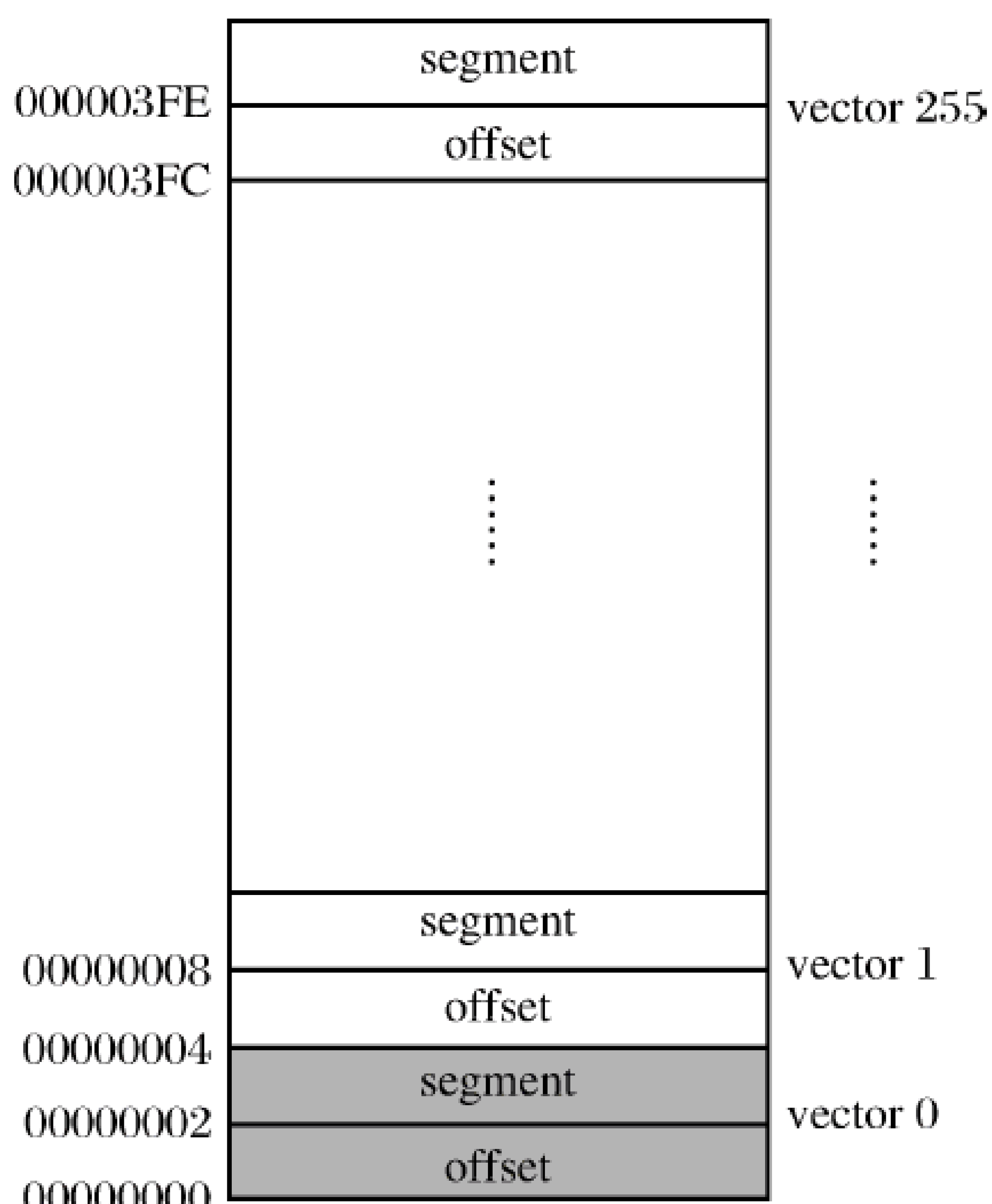


图 20－21　实模式下的 IVT

20.2　注入技术

注入技术也是一种传统技术，其核心思想就是将具有特定功能但却与目标进程不相关的模块或代码段“加载”到目标进程的地址空间中，使之成为目标进程的一部分，注入模块能够访问目标进程其他模块中的资源，或者直接执行被注入的代码段，甚至可对目标模块进行挂钩，可以说是目标进程中的“余则成”。有多种注入实现技术，这些被注入的代码段也常被称为 ShellCode。

20.2.1　APC 注入方式

APC（Asynchronous Procedure Call，异步过程调用）是与线程相关的，每个线程都有自己的 APC 队列。当线程切换或处于可变等待状态时，线程的 APC 队列就会被调度执行。这种机制也可以被用来实现注入技术，具体做法就是将模块加载函数封装进 APC 中，并将该 APC 投递到目标进程任意线程的 APC 队列中等待执行。

但要注意的是 APC 挂入队列后不会立即执行，而要等到被挂入队列所在线程处于可变等待状态或被调度切换时才会被执行。虽然如此，这对于大多数注入场景已经够用了。

以下 5 个函数能够使线程进入可变等待状态：SleepEx、WaitForSingleObjectEx、WaitForMultipleObjectsEx、SignalObjectAndWait、MsgWaitForMultipleObjectsEx。

APC 注入的核心步骤非常简单，如下所示：

（1）**打开目标进程**：使用 Windows API OpenProcess 可以实现。

(2) **在目标进程中分配被注入模块的内存空间**:通过 VirtualAllocEx 方法实现,设置的内存空间属性为 PAGE_EXECUTE_READWRITE,其参数为 OpenProcess 打开的目标进程句柄。

(3) **写入需要注入的模块路径全名**:使用 WriteProcessMemory 方法实现,其参数也是目标进程句柄和刚刚分配的内存空间地址。

(4) **创建线程快照并查找目标进程中的线程**:使用 CreateToolhelp32Snapshot 方法将系统中的所有线程枚举出来,通过判断线程所属的进程 ID 查找出目标进程中的线程。

(5) **向目标线程添加 APC**:使用 QueueUserAPC 方法添加 APC,其参数分别为:

➢ LoadLibraryA,将该模块加载函数的地址作为 APC 函数指针;

➢目标线程句柄;

➢步骤(2)中创建的 APC 函数的参数内存块地址。

20.2.2 注册表注入方式

当一个新的 DLL 被进程加载时,Windows 系统中的 user32. dll 模块会调用 LoadLibrary 方法加载所有由 AppInit_DLLs 指定的 DLL 模块。AppInit_DLLs 是注册表的系统设置项,可以为任一个进程调用一个 DLL 列表,在注册表中的具体位置如图 20－22 所示。

SoftwarePr
SPP
Superfetch
Svchost
SystemRest
Terminal Se
Time Zones
Tracing
UnattendSe
Userinstallat
WbemPerf
Windows
Winlogon
Winsat
WinSATAPI
WUDF
Windows Photo Vi

名称	类型	数据
IconServiceLib	REG_SZ	IconCodecService.dll
DdeSendTimeout	REG_DWORD	0x00000000 (0)
DesktopHeapLog...	REG_DWORD	0x00000001 (1)
GDIProcessHandl...	REG_DWORD	0x00002710 (10000)
ShutdownWarnin...	REG_DWORD	0xFFFFFFFF (-1)
USERNestedWin...	REG_DWORD	0x00000032 (50)
USERPostMessag...	REG_DWORD	0x00002710 (10000)
USERProcessHan...	REG_DWORD	0x00002710 (10000)
(Default)	REG_SZ	mnmsrvc
DeviceNotSelect...	REG_SZ	15
Spooler	REG_SZ	yes
TransmissionRetr...	REG_SZ	90
AppInit_DLLs	REG_SZ	
LoadAppInit_DLLs	REG_DWORD	0x00000000 (0)

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows

图 20－22 AppInit_DLLs 在注册表中的位置

这种注入方式是通过 user32. dll 的入口函数实现的。在 user32. dll 的入口函数 DllMain 中,当遇到进程加载事件(DLL_PROCESS_ATTACH)时,会检查这个注册表目录,如果存在需要预加载的模块则加载它们,也就是说 user32. dll 加载了这些模块。但这种注入方式的弊端是目标进程必须引用了 user32. dll。

20.2.3　远程线程注入方式

CreateRemoteThread 方法是由 kernel32. dll 模块导出的 Windows API 之一（如图 20 – 23 所示），其作用是在目标进程中创建一个线程，而目标进程可以是当前进程，也可以是系统中的其他进程，但不能跨主机创建，且同主机的两个不同进程必须在一个登录会话中。

CreateRemoteThread 方法多用于远程注入技术，即当前进程向其他进程注入模块。这里所说的远程不是跨主机或者跨会话，而是跨进程。可见在操作系统中，由于其用户态进程空间的隔离性，即使是在同一系统的同一会话中，不同进程也被视作“远程”了。这里还要提一点，CreateRemoteThread 只适用于 Windows Vista 以下的版本，Windows Vista 及以上版本要改用 CreateRemoteThreadEx 方法。

File: kernel32.dll
Dos 头部
Nt 头部
文件头部
可选头部
数据目录 [x]
段头部 [x]
输出目录

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	0009F4D0	000A1F02	000A0A94	000A3654
(nFunctions)	Dword	Word	Dword	szAnsi
000000AB	0004C840	00AA	000A4441	CreateRemoteThread
000000AC	000AA376	00AB	000A4454	CreateRemoteThreadEx

图 20 – 23　kernel32. dll 导出 CreateRemoteThread 和 CreateRemoteThreadEx

CreateRemoteThread 的参数如下所示：

```
HANDLE
WINAPI
CreateRemoteThread(
    _In_ HANDLE hProcess,                                    //远程线程的句柄
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,       //安全属性
    _In_ SIZE_T dwStackSize,                                 //栈大小
    _In_ LPTHREAD_START_ROUTINE lpStartAddress,              //进程处理函数
    _In_opt_ LPVOID lpParameter,                             //进程参数
    _In_ DWORD dwCreationFlags,                              //默认创建后的状态
    _Out_opt_ LPDWORD lpThreadId                             //所创建的线程的 ID
    );
```

使用该方法需要经过如下几个步骤：

（1）**根据进程 ID 打开目标进程**：使用 OpenProcess 方法得到进程句柄。

（2）**根据进程句柄在目标进程中申请内存**：使用 VirtualAllocEx 方法分配内存。

（3）**在目标进程中对刚刚申请的内存写入所需参数（需加载的 DLL 的完整路径）**：可以采用 WriteProcessMemory 方法实现。

（4）**获取 LoadLibrary 方法的模块加载地址**：可使用 GetProcAddress 方法实现。LoadLibrary 是由 kernel32. dll 导出的函数，而 kernel32. dll 的加载地址一般是固定的，因此使用 GetProcAddress 方法获得的 LoadLibrary 函数地址在各个进程中都是一样的。

（5）**启动远程线程**：使用 CreateRemoteThread。

20.2.4 浏览器辅助对象方式

1. 浏览器劫持

浏览器劫持(Browser Hijack)是一种不同于普通病毒感染途径的网络攻击手段,它的渗透途径很多,目前最常见的方式一般是利用浏览器辅助对象(Browser Helper Object,BHO)、浏览器 DLL 插件、挂钩、WinSock LSP 等载体达到对用户浏览器进行恶意篡改和劫持的目的。其中,利用 BHO 是浏览器劫持的重要手段,甚至是主要手段。这些载体可以直接寄生于浏览器的模块里,成为浏览器的一部分,进而直接操纵浏览器的行为。我们常见的打开 IE 却链接到非正常网站就是浏览器劫持的例子。

2. 浏览器辅助对象

浏览器辅助对象(BHO)是微软推出的作为浏览器对第三方程序开发的交互接口的标准,通过这个接口就可以编写代码来拓展浏览器(IE)的功能并获取浏览器的行为(前进、后退、当前页面)等。当然,剑都是双刃的,这种看似人畜无害的技术也给了恶意程序可乘之机。借助 BHO,可以实现一个在每次启动浏览器时都加载的 COM 对象,这个 COM 对象运行于浏览器的进程空间中,完全可以实现恶意逻辑。

由于 BHO 依托于浏览器主窗口,因此 BHO 与浏览器实例的生命周期是一致的。符合 BHO 接口标准的程序代码被封装成 DLL 的形式并在注册表里注册为 COM 对象,还要在 BHO 接口的注册表入口处进行组件注册,以后每次 IE 启动时都会通过此处描述的注册信息调用加载这个 DLL 文件,该文件就因此成为 IE 的一个构成模块(BHO 组件),与 IE 共享一个运行周期,直到 IE 被关闭。

IE 启动时会加载任何 BHO 组件,这些组件直接进入 IE 内存区域,而 IE 则成为它们的父进程和载体,从此 IE 的每一个事件都会通过 IUnknown 接口传递到 BHO 用于提供交互的 IObjectWithSite 接口,这是 BHO 实现与 IE 交互的入口函数。当然,BHO 无法窥知浏览的内容,例如页面的内容等。

由于 BHO 是借助浏览器启动了恶意代码 DLL 的加载,因此这种方式也被归类为注入技术。图 20－24 显示了 Windows 7 系统下的 BHO 插件。

名称	类型	模块路径	文件厂商	CLSID
&使用&迅雷下载	IE右键菜单	c:\program files (x86)\thunder network\thunder9\bho\geturl.htm		
&使用&迅雷下载全部链接	IE右键菜单	c:\program files (x86)\thunder network\thunder9\bho\getallurl.htm		
&使用&迅雷离线下载	IE右键菜单	c:\program files (x86)\thunder network\thunder9\bho\offlinedownload.htm		
Adobe PDF Link Helper	BHO插件	C:\Program Files (x86)\Common Files\Adobe\Acrobat\ActiveX\AcroIEHelperShim.dll	Adobe Systems Incorp...	{18DF08
Skype for Business Bro...	BHO插件	C:\Program Files (x86)\Microsoft Office\Office15\OCHelper.dll	Microsoft Corporation	{31D09E
WebProtect	BHO插件	C:\Program Files\CMBCHINA\WebProtect\WebProtect.dll	China Merchants Bank	{53763D
Office Document Cach...	BHO插件	C:\Program Files (x86)\Microsoft Office\root\Office16\URLREDIR.DLL	Microsoft Corporation	{B4F3A8
SafeMon Class	BHO插件	C:\Program Files (x86)\360\360Safe\safemon\safemon.dll	360.cn	{B69F34
AccountProtectBHO Class	BHO插件	C:\Users\ZNV\AppData\Roaming\Tencent\QQ\QQAntiPhishing\AccountProtect.dll	Tencent	{DDD36
KingdeePLMClient.Upda...	ActiveX插件	C:\Windows\Downloaded Program Files\kingdeeplmclient.dll	Kingdee	{00E9C0

图 20－24　某 Windows 7 系统下的 BHO 插件

3. BHO 的使用

BHO 是个 COM 进程服务,必须注册于注册表的某一键下,IE 和 Explorer 将查询这个键

并加载键下所有的对象,BHO 所在的注册表键如图 20-25 所示。若想使用 BHO,首先需要生成一个 DLL 文件,并在注册表下注册一个子键,然后在 HKEY_CLASSES_ROOT\CLSID 路径下注册一个 CLSID 子键并再建立一个名为 InprocServer32 的子健,默认值设置为该 DLL 文件的完整路径。

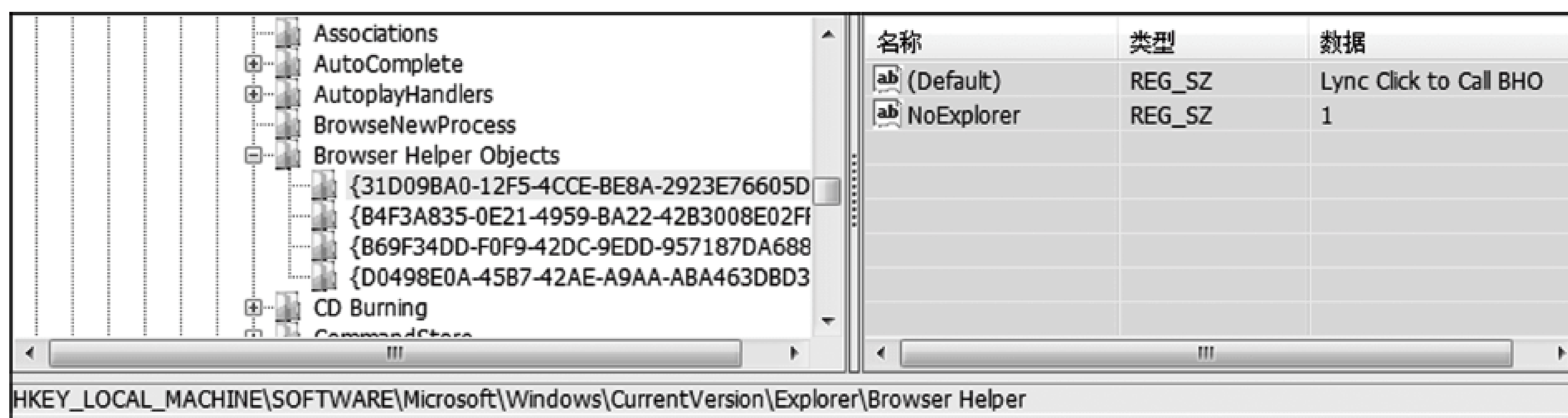


图 20-25　BHO 所在的注册表键

IE 与 BHO 关联的步骤是这样的:

(1) 打开 IE 窗口时,先寻找上述注册表键下的 CLSID,这些 CLSID 都对应着相应的 BHO 插件,然后根据这个 CLSID 到 HKEY_CLASSES_ROOT 下的 CLSID 里找到此插件的信息,例如文件位置等。

(2) IE 根据找到的 CLSID 信息创建 BHO 对象,并且查找 IObjectWithSite 接口(这个接口只有 SetSite 和 GetSite 两个方法)。

(3) IE 把浏览器插件 IWebBrowser2 传到 BHO 的 SetSite 方法中,用户在此方法中可挂钩自己的事件处理例程。

(4) 窗口关闭时,IE 将 NULL 参数传入 BHO 的 SetSite 方法以用于卸载挂钩的事件处理例程。

20.2.5　DLL 劫持

DLL 劫持也属于一种变相的注入技术,其核心思想是“注入”一个伪造的必选模块以实现目标进程数据的截获。

在 PE 文件的导入表中只包含了 DLL 文件的名称而没有包含其路径名,模块加载程序在加载动态库的时候必须在磁盘上搜索 DLL 文件。但是搜索 DLL 文件是有优先顺序的:

(1) 加载程序首先会尝试从当前程序所在的目录加载 DLL 文件;

(2) 如果上一步没找到,则在 Windows 系统目录中查找;

(3) 如果还是没找到,则在环境变量列出的各个目录中查找;

(4) 如果上述三个步骤都没有找到,则搜索加载失败。

这一特点正可以被利用。比如我们先伪造一个系统同名 DLL 文件(例如 kernel32.dll)并提供相同的导出表,表中每个导出函数转向真正的系统 DLL 文件(kernel32.dll)。如此一来程序调用系统 DLL 文件时会先搜索调用当前 EXE 文件所在目录下的系统同名 DLL 文件,也就是这个伪造的 DLL 文件(正常情况下在当前目录下是搜索不到的,只能跳到第二步在

Windows 系统目录中查找,由于真正的系统 DLL 文件在第二步的目录中,因此此时可以搜索成功)。但没想到在第一步中就搜索到了且导出表相同,这实在是令人太惊喜、太意外了,自然也就加载成功了。可以在这个伪造的 DLL 文件中完成挂钩所要完成的功能,再转到 Windows 系统目录中去执行真正的 DLL 文件。我们称这个过程为 DLL 劫持。

DLL 劫持是一种较为笨重的办法,其弊端在于要实现与原目标 DLL 文件相同的导出表,且要保持两者的参数数量和调用约定的一致性,这要求攻击者对目标 DLL 文件非常熟悉,从而大大限制了使用的便捷性和通用性。

20.2.6 PE 感染

PE 文件无论是在内存中还是在磁盘中,都要按照一定的字节数对齐。例如在 64 位系统下的内存和磁盘中,32 位的 PE 文件均以 4 KB 为步长对齐(如图 20 – 26 中的 SectionAlignment 和 FileAlignment 域所示)。

SIPGateModule.dll

Member	Offset	Size	Value
Magic	00000118	Word	010B
MajorLinkerVersion	0000011A	Byte	06
MinorLinkerVersion	0000011B	Byte	00
SizeOfCode	0000011C	Dword	002E1000
SizeOfInitializedData	00000120	Dword	001D5000
SizeOfUninitializedData	00000124	Dword	00000000
AddressOfEntryPoint	00000128	Dword	00296981
BaseOfCode	0000012C	Dword	00001000
BaseOfData	00000130	Dword	002E2000
ImageBase	00000134	Dword	10000000
SectionAlignment	00000138	Dword	00001000
FileAlignment	0000013C	Dword	00001000

图 20 – 26 PE 文件的可选头

但是 PE 头、代码段和数据段等不可能正好占用 4 KB 或者 4 KB 的整数倍大小的空间,都会或多或少留有一些缝隙,而这些缝隙中就可以插入少量恶意代码,甚至在 PE 文件的尾部可以通过扩展空间的方式插入大量恶意代码,而在 PE 的入口地址处将入口地址改为指向这些代码,这种注入方式就叫作 PE 感染。

如图 20 – 27 所示,PE 文件的入口地址被修改为已插入代码的基址。当 PE 文件被加载时,首先就会执行被插入的代码,待完成插入代码的执行后再跳转回正常的入口地址(即修改 PE 文件之前 AddressOfEntryPoint 指向的地址),从而完成恶意代码的注入操作。这里是采用在尾部增加新的 Section(区段)的方式来实现恶意代码注入的。尾部插入的方式不受 PE 文件各区段之间“缝隙”大小的限制,具有较大的灵活性,但要注意增加新的 Section 后也

要修改文件的大小,包括可选头中的 SizeOfImage 域和 PE 头中的 NumberOfSections 域等成员变量。

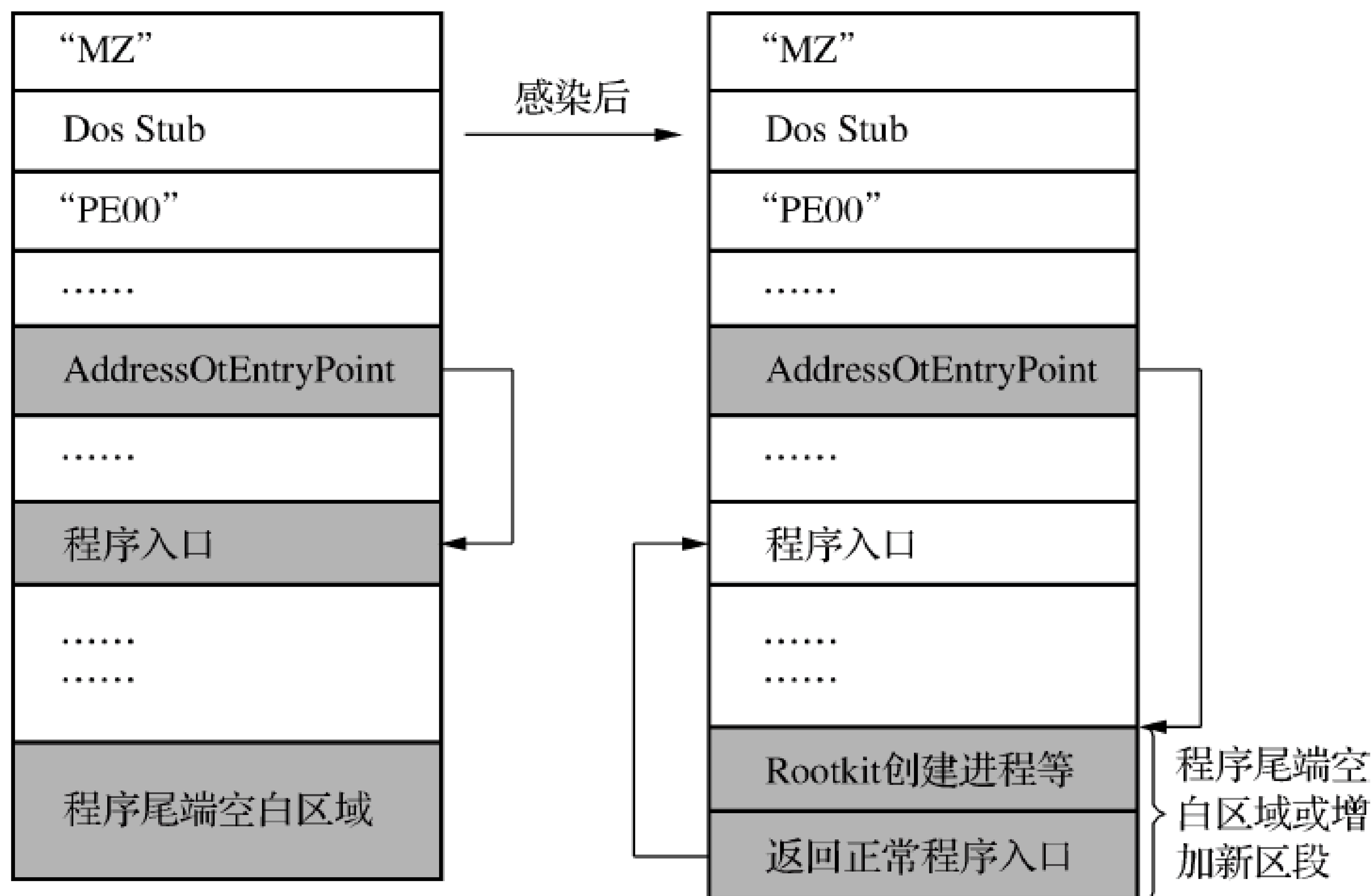


图 20－27　一种基于尾部插入的 PE 感染方式

20.3　过滤驱动技术

在 Windows 中,对于 IRP 的过滤一般是采用过滤型驱动的方式进行的。对于一些规模较大且深度较深的协议栈,例如文件系统驱动或网络协议驱动,Windows 都有专门的过滤框架供第三方进行报文/数据过滤。其中文件系统驱动的过滤框架是 FltMgr,而网络协议驱动的过滤框架则是 WFP。这两个框架是操作系统过滤驱动的集大成者,在前文中都有介绍,本节不再赘述。

在驱动程序中还有一种过滤技术,就是对 IRP 的功能函数(派遣函数)和快速 I/O 函数进行 Inline Hook,或直接替换功能函数数组的函数指针为其他模块函数的地址,而从技术上看这又属于类似 EAT Hook 或 Inline Hook 的方式了,挂钩点和挂钩方式也都大同小异,在这里也不赘述。

过滤驱动技术主要是以过滤设备对象的方式堆叠(由 IoAttachDevice 方法实现堆叠)在驱动设备栈中,并以此来过滤 IRP,I/O 管理器向设备发送的 IRP 会被中间的过滤驱动截获,过滤驱动可以选择截获阻断这个 IRP 而直接返回,也可以篡改这个 IRP 再下传,具有很大的灵活性。但这种方式会对驱动设备栈的深度和 IRP 堆栈的深度造成影响,因为其本质就是在驱动栈中内嵌了一层,这一层驱动也具备各种主副功能码的派遣函数,就像是一层起着过滤作用的功能驱动。例如我们在 PC 上登录网上银行的 U 盾时,往往感觉鼠标和键盘会发生暂时的卡顿现象,就是因为 U 盾程序在启动和激活键盘和鼠标的过滤型驱动而造成的。

图 20－28 中显示了 Windows 7 系统下的各种过滤型驱动模块,除了微软自身堆叠的过

滤设备,包括一些系统保护软件在内也会在驱动栈中堆叠过滤设备,例如 360NsiFilter 就是防攻击软件 360AntiHacker64. sys 堆叠的设备。

系统回调 | 过滤驱动 | DPC定时器 | 工作队列线程 | Hal | Wdf | 文件系统 | 系统调试 | 对象劫持 | 直接IO | GDT

类型	驱动对象名	驱动路径	设备	设备名	宿主驱动对象名	文件厂商
File	\FileSystem\FltMgr	C:\Windows\System32\drivers\FLTMGR.SYS	0xFFFFFA8007422DE0		\FileSystem\Ntfs	Microsoft Corporation
Disk	\Driver\partmgr	C:\Windows\System32\drivers\partmgr.sys	0xFFFFFA80074008D0		\Driver\Disk	Microsoft Corporation
Raw	\FileSystem\FltMgr	C:\Windows\System32\drivers\FLTMGR.SYS	0xFFFFFA80072C28C0		\FileSystem\RAW	Microsoft Corporation
Raw	\FileSystem\FltMgr	C:\Windows\System32\drivers\FLTMGR.SYS	0xFFFFFA80072C2DB0		\FileSystem\RAW	Microsoft Corporation
Volume	\Driver\fvevol	C:\Windows\System32\DRIVERS\fvevol.sys	0xFFFFFA800740BA40		\Driver\volmgr	Microsoft Corporation
Volume	\Driver\volsnap	C:\Windows\system32\drivers\volsnap.sys	0xFFFFFA800751D040		\Driver\fvevol	Microsoft Corporation
I8042prt	\Driver\SynTP	C:\Windows\system32\DRIVERS\SynTP.sys	0xFFFFFA800782EAD0		\Driver\i8042prt	Synaptics Incorporated
I8042prt	\Driver\mouclass	C:\Windows\system32\DRIVERS\mouclass.sys	0xFFFFFA80077A0AD0	PointerClass0	\Driver\SynTP	Microsoft Corporation
I8042prt	\Driver\SynTP	C:\Windows\system32\DRIVERS\SynTP.sys	0xFFFFFA800769BE10		\Driver\i8042prt	Synaptics Incorporated
I8042prt	\Driver\kbdclass	C:\Windows\system32\DRIVERS\kbdclass.sys	0xFFFFFA80077A5CE0	KeyboardClass0	\Driver\SynTP	Microsoft Corporation
Nsiproxy	\Driver\360AntiHacker	C:\Windows\System32\Drivers\360AntiHacker64.sys	0xFFFFFA800758A060	360NsiFilter	\Driver\nsiproxy	360.cn
PnpManager	\Driver\volmgr	C:\Windows\system32\drivers\volmgr.sys	0xFFFFFA8007215CE0	VolMgrControl	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\vdrvroot	C:\Windows\system32\drivers\vdrvroot.sys	0xFFFFFA80072123B0		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\dtliteusbbus	C:\Windows\system32\DRIVERS\dtliteusbbus.sys	0xFFFFFA8007798170		\Driver\PnpManager	Disc Soft Ltd
PnpManager	\Driver\umbus	C:\Windows\system32\DRIVERS\umbus.sys	0xFFFFFA8007797C70		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\swenum	C:\Windows\system32\drivers\swenum.sys	0xFFFFFA8007792040		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\psadd	C:\Windows\system32\DRIVERS\psadd.sys	0xFFFFFA800778B4D0	PsaDD0	\Driver\PnpManager	Lenovo (United States)
PnpManager	\Driver\dtlitescsibus	C:\Windows\system32\DRIVERS\dtlitescsibus.sys	0xFFFFFA800778B040		\Driver\PnpManager	Disc Soft Ltd
PnpManager	\Driver\TermDD	C:\Windows\system32\drivers\termdd.sys	0xFFFFFA8007782DA0		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\mouclass	C:\Windows\system32\DRIVERS\mouclass.sys	0xFFFFFA8007788CF0	PointerClass1	\Driver\TermDD	Microsoft Corporation
PnpManager	\Driver\TermDD	C:\Windows\system32\drivers\termdd.sys	0xFFFFFA8007782040		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\kbdclass	C:\Windows\system32\DRIVERS\kbdclass.sys	0xFFFFFA8007782960	KeyboardClass1	\Driver\TermDD	Microsoft Corporation
PnpManager	\Driver\rdpbus	C:\Windows\system32\DRIVERS\rdpbus.sys	0xFFFFFA800777D950	RdpBus	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\msloop	C:\Windows\system32\DRIVERS\loop.sys	0xFFFFFA8007786050	NDMP15	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\RasSstp	C:\Windows\system32\DRIVERS\rassstp.sys	0xFFFFFA8007784050	NDMP14	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\PptpMiniport	C:\Windows\system32\DRIVERS\raspptp.sys	0xFFFFFA800777F050	NDMP13	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\RasPppoe	C:\Windows\system32\DRIVERS\raspppoe.sys	0xFFFFFA8007771050	NDMP12	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\NdisWan	C:\Windows\system32\DRIVERS\ndiswan.sys	0xFFFFFA800776C050	NDMP11	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\NdisWan	C:\Windows\system32\DRIVERS\ndiswan.sys	0xFFFFFA80076F6050	NDMP10	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\NdisWan	C:\Windows\system32\DRIVERS\ndiswan.sys	0xFFFFFA800770E050	NDMP9	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\Rasl2tp	C:\Windows\system32\DRIVERS\rasl2tp.sys	0xFFFFFA8007715050	NDMP8	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\RasAgileVpn	C:\Windows\system32\DRIVERS\AgileVpn.sys	0xFFFFFA8007727050	NDMP7	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\mssmbios	C:\Windows\system32\drivers\mssmbios.sys	0xFFFFFA8007760720		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\Compbatt	C:\Windows\system32\DRIVERS\compbatt.sys	0xFFFFFA8007214490	CompositeBattery	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\CompositeBus	C:\Windows\system32\DRIVERS\CompositeBus.sys	0xFFFFFA8007760040		\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\blbdrive	C:\Windows\system32\DRIVERS\blbdrive.sys	0xFFFFFA800758B3C0	BlbControl	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\ACPI_HAL		0xFFFFFA8006AE66E0		\Driver\PnpManager	
PnpManager	\Driver\tunnel	C:\Windows\system32\DRIVERS\tunnel.sys	0xFFFFFA8007638050	NDMP4	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\tunnel	C:\Windows\system32\DRIVERS\tunnel.sys	0xFFFFFA8007635050	NDMP3	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\tunnel	C:\Windows\system32\DRIVERS\tunnel.sys	0xFFFFFA8007631050	NDMP2	\Driver\PnpManager	Microsoft Corporation
PnpManager	\Driver\tunnel	C:\Windows\system32\DRIVERS\tunnel.sys	0xFFFFFA800762E050	NDMP1	\Driver\PnpManager	Microsoft Corporation

图 20－28　某 Windows 7 系统下的过滤型驱动模块

同时,过滤型驱动还具有自己的特点,包括:

(1) **无需电源管理**。由于过滤型驱动并非真正的设备驱动,而且它们不需要直接控制硬件设备,因此过滤型驱动并不需要接收电源请求 IRP_MJ_POWER,电源管理 IRP 将越过过滤型驱动而被直接发送到底层设备堆栈中。不过在一些特殊情况下,文件系统过滤驱动有可能会影响到电源管理。

(2) **过滤型驱动都是非 WDM 驱动**。文件系统过滤驱动并不是 WDM 驱动,WDM 驱动模型仅适用于设备驱动。

(3) **过滤型驱动都没有 AddDevice 或 StartIo 例程**。由于过滤型驱动并非设备驱动且它们并不直接控制硬件设备,因此它们没有 AddDevice 或 StartIo 例程,设备的堆叠一般由 IoAttachDevice 方法完成。

20.4　Bootkit 技术

Bootkit 是更高级的 Rootkit 手段,也是更底层的 Rootkit。Bootkit 分为软件 Bootkit 和硬件 Bootkit。软件 Bootkit 通过感染启动文件 NTLDR 或 boot. ini 实现了先于操作系统内核的启动;硬件 Bootkit 通过感染 MBR(主引导扇区)甚至 BIOS 实现了内核检测的绕过和隐身启动。由于 MBR 比 Windows 内核加载得更早,且 Bootkit 的运行无文件、无进程、无模块,只存在于运行的内存和操作系统管辖范围之外的磁盘扇区空间里,基于操作系统的防护软件对

Bootkit 鞭长莫及,没有任何作用,因此 Bootkit 比其他普通的 Rootkit 更难防御,更隐蔽,也更顽固,即使重装了操作系统,Bootkit 依然存在。

一般来说,所有比 Windows 内核更早加载的 Rootkit 都可归类为 Bootkit,例如 BIOS Rootkit、VBootkit、SMM Rootkit 等,但目前最为常见的还是基于感染 MBR 的 Bootkit。

1. SMM Rootkit

CPU 的操作模式一般分为实模式、保护模式、V86 模式和处理器系统管理模式。处理器系统管理模式(System Management Mode,SMM)面向系统固件,主要用于电源管理、系统安全等方面的操作。在 SMM 下,处理器通过系统管理中断(System Management Interrupt,SMI)切换到一个独立的地址空间,同时保存当前运行的程序或任务的上下文。从 SMM 返回时,处理器返回 SMI 前的工作模式。

SMM 不常见,只有当 CPU 中 SMM 中断的引脚被触发时(一般是由外部的 APIC 引起),CPU 才会进入 SMM。在 SMM 下操作系统、设备中断等都会处于失效状态,因此 SMM 具有很高的权限。SMM 对于操作系统是透明的,操作系统无法感知 SMM 的切换和退出。所有的 SMM 处理例程只能在系统管理内存(System Management RAM,SMRAM)空间中运行,而 SMM 处理例程也只能由系统固件实现。

CPU 处于 SMM 时会执行 SMI 处理例程:首先触发 SMI,继而 CPU 将当前状态存储于 SMRAM 中,再执行位于这个 SMRAM 中的 SMI 处理例程,最后 CPU 遇到 RSM 指令而返回。由于 SMI 处于 SMM,因此可以随意访问内核数据。这里要强调的是,普通的软件是不能触发 SMI 的,能触发的要么是 CPU 的 SMI 引脚信号,要么是高级可编程中断控制器(APIC)总线上的 SMI 消息。

Bootkit 对 SMI 的利用有下列几种方式:

- 将 SMRAM Controller 中的 D_OPEN 标志位置 1,使得非 SMM 的代码(Bootkit)可以访问 SMRAM。
- 修改 SMRAM 的 SMI 区域,使之指向 Bootkit 所属的代码区域,当触发了 SMI 后会执行到 Bootkit。
- 在 SMI 分配表中添加新的 SMI 处理例程,这个处理例程指向 Bootkit。

2. MBR Bootkit

CPU 加电后首先进入实模式以进行相应的初始化,包括 GDT 等数据结构都是在实模式下完成初始化的(实模式下的寻址空间为 1 MB),完成后切换到保护模式,X86 系统中的大部分代码都是运行在保护模式下的(32 位保护模式下的寻址空间为 4 GB)。

基于 MBR 的 Bootkit 就是运行于这两种模式下的 Rootkit,因此这种 Bootkit 要具有实模式执行和保护模式执行这两部分代码,并且要实现实模式到保护模式的切换。Bootkit 在启动时先加载自己的代码,再加载操作系统,以此方式为自己建立了有利的启动环境。

我们先来回顾一下操作系统的启动流程,如图 20－29 所示。

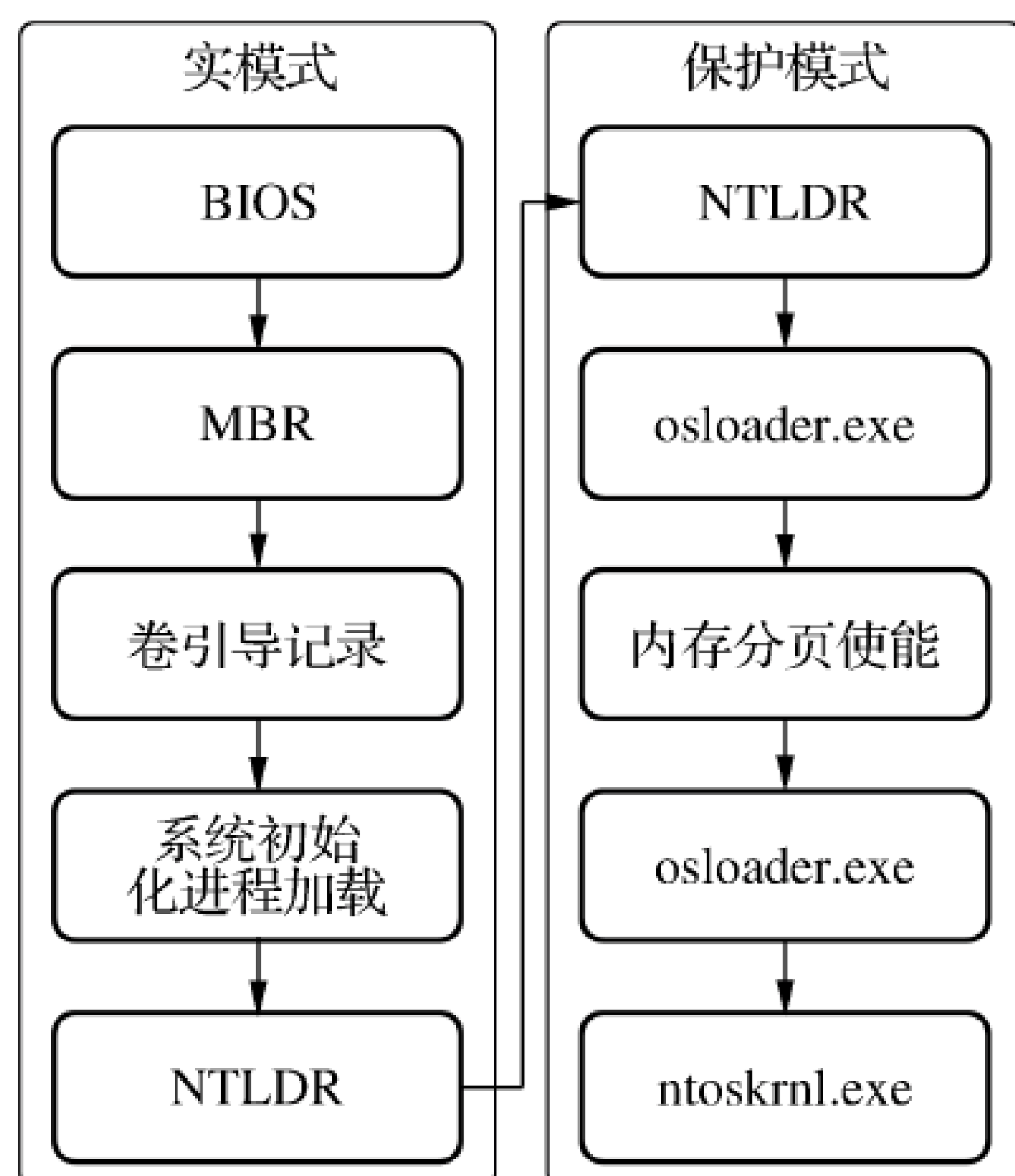


图 20－29　BIOS + MBR 方式的操作系统启动流程

在 Windows 系统中比较常用的是 Windows NT 5. X 系列和 Windows NT 6. X 内核系列，NT 5. X 内核涵盖 Windows XP、Windows Server 2003 等系统；NT 6. X 系列内核则涵盖 Window Vista、Windows 7、Windows Server 2008 及以上的系统，这两种内核遵循不同的引导机制，如图 20－30 所示。

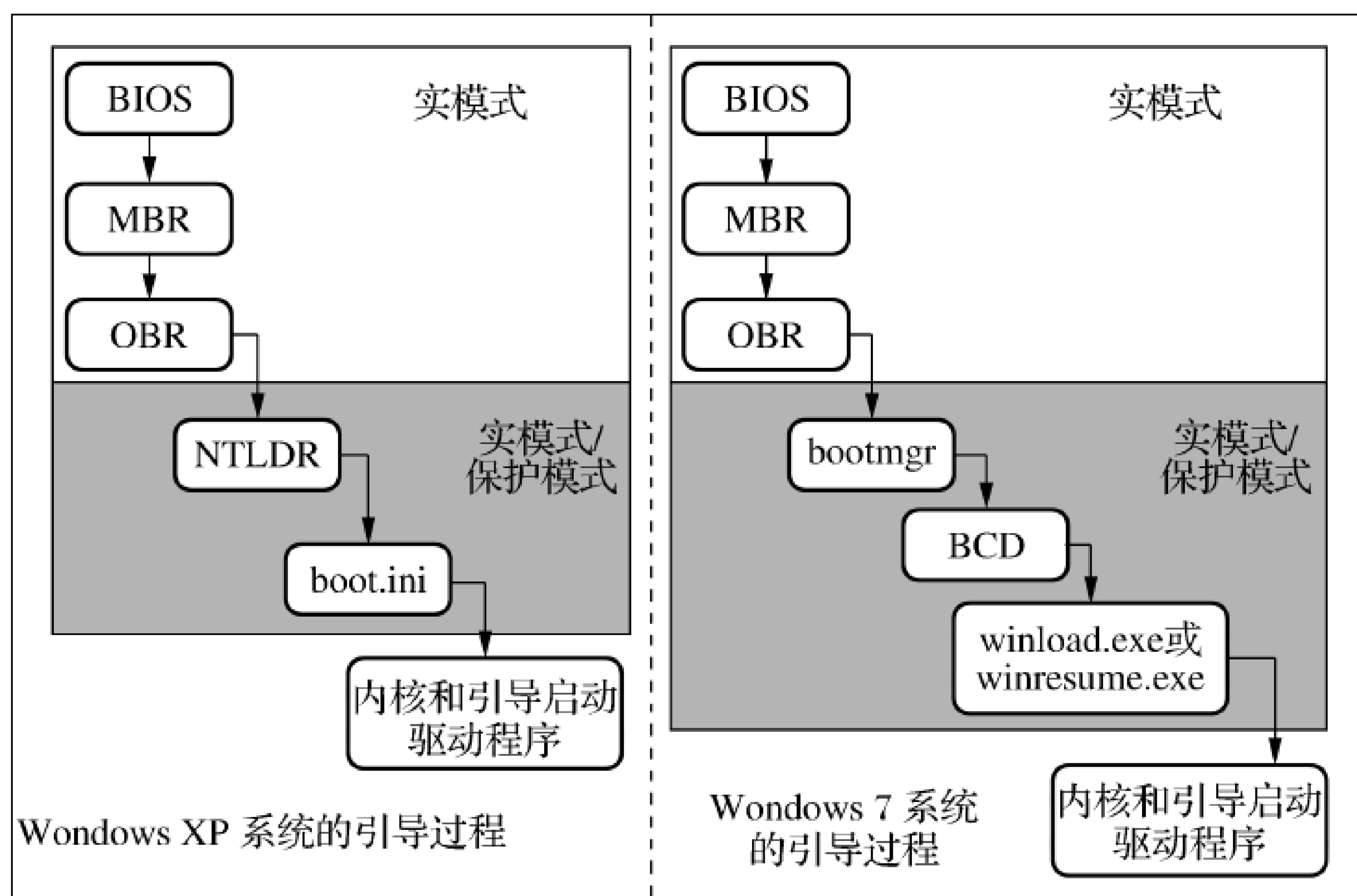


图 20－30　NT 5.0 内核与 NT 6.0 内核的引导过程对比

我们以 NT 5.0 内核的系统为例。MBR 是开机后 BIOS 自检所加载的第一个扇区，BIOS + MBR 方式也是 5.0 内核采用的启动方式，开机后其流程是这样的：

（1）BIOS 加电自检（Power On Self Test，POST），其实模式下的内存地址为 FFFF:0000。

（2）将硬盘第一个扇区（0 头 0 道 1 扇区，也就是 MBR）读入内存地址 0000:7C00 处，读取扇区使用的是 int 0x13 中断指令。

（3）检查 0000:7DFE 处是否等于 0xAA55，若不相等则转去尝试其他启动介质，如果没

有其他启动介质则显示“No ROM BASIC”,最后执行 BSOD,流程结束。

(4) 若相等则跳转到 0000:7C00 处执行 MBR 中的程序,以查找活动分区。

(5) 在主分区表中搜索活动标志的分区,如果发现没有活动分区或有不止一个活动分区时,引导分区停止启动(只能有一个活动分区被标记为可引导分区)。

(6) 将活动分区的第一个扇区 VBR(Volume Boot Record,卷引导记录)读入内存地址 0000:7C00 处。

(7) 检查 0000:7DFE 处是否等于 0xAA55,若不相等则显示“Missing Operating System”,然后停止启动或尝试其他介质启动。

(8) 跳转到 0000:7C00 处继续执行特定系统的启动程序加载器(Inital Program Loader,IPL)。IPL 的任务是在磁盘上定位 NTLDR,并将其读入内存。

(9) 通过 NTLDR 启动 Windows 系统,进入保护模式。

这其中的步骤(1)~(4)都是由 BIOS 来引导的,步骤(5)~(9)则是由 MBR 中的引导程序来引导的。基于 MBR 的 Bootkit 就是将 MBR 中的引导程序替换为自己的恶意代码,使系统不再去查找 VBR,这样系统一启动就加载 Bootkit 代码了。

MBR Bootkit 根据代码的运行环境可以分为 MBR 阶段、后实模式阶段、保护模式阶段、系统内核阶段和应用程序阶段。每个 Bootkit 可能涵盖其中的部分或全部阶段,且每个阶段都相对独立。

3. 基于可信计算的 Bootkit 检测

Bootkit 一般具有较强的隐蔽性和自我防护措施,可以通过挂钩磁盘驱动的读写例程以欺骗杀毒软件,使感染的磁盘无法正常修复,并使之无法获取真正的主引导扇区的代码。有的 Bootkit 也可以检测和应付杀毒软件,采用真实的主引导区代码进行欺骗和伪装。MBR Bootkit 会在内存中开辟一个隐蔽空间,用于加载执行自身的恶意代码,并常驻在系统内存中。Bootkit 会尽量避免在磁盘中产生程序文件,因而更加难以检测。

基于可信计算技术的 Bootkit 检测是一种较为精确的检测方式。系统中安装了可信计算基(TCB)检测部件,之后提取系统启动关键模块的 MD5 校验值并对数据进行度量,获取系统可信度的度量值并校验,以实现系统启动流程的动态监测。该检测方法能够识别未知的 Bootkit 攻击。

图 20-31 描述的是嵌入了 TPM(可信平台模块)可信计算基运行模块的操作系统的引导和启动过程。在可信计算 3.0 体系中,基于主动免疫的可信计算技术可以动态地识别计算机系统中的“自我”和“非我”,并且免装检测软件实体,实现了更好的隐蔽性和安全性。

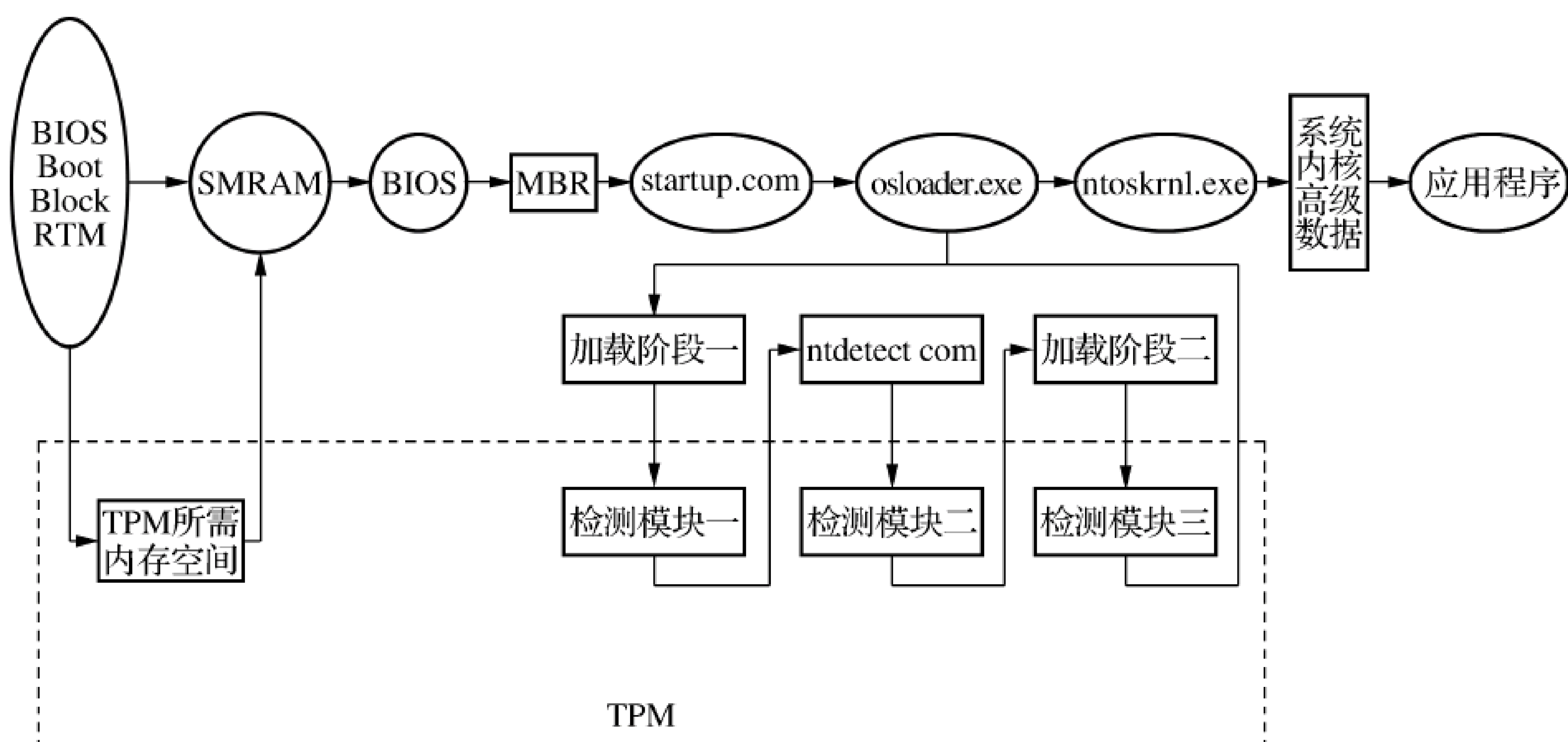


图 20－31　基于可信链传递的内核模块校验流程

本章小结

本章介绍了 Rootkit 和 Bootkit 的相关技术。

Rootkit 最常用的手段有：挂钩、注入和过滤，这三种都是非常传统的、经历过攻防双方的演进发展和对抗锤炼的手段。

挂钩技术的基本思想是截获和改变代码的处理流程，包括了内核态挂钩与用户态挂钩两个方面：内核态挂钩有 SSDT Hook、IDT Hook、GDT Hook、MSR Hook、Object Hook、DKOM 等方式；用户态挂钩有 IAT/EAT Hook、消息报文 Hook 等方式。而最传统的 Inline Hook 既可用于内核态也可用于用户态，至于 IVT Hook 更是跳出三界之外，不在保护模式中了。

注入技术则着重于实现恶意模块向正常进程的渗透，将一个“非我”的模块加载到“自我”的进程空间中，手段包括 APC 注入、注册表注入、远程线程注入、BHO 方式、DLL 劫持和 PE 感染等。这些手段中有的比较具有普适性和通用性，有的则受限于不少条件。

不过随着 Windows 内存安全手段的加强，无论是挂钩还是注入，其实现门槛都大大地提高了。而当前 Windows 提出的驱动签名技术，更是给过滤型驱动带来了很大的挑战。

Bootkit 是一种更为前期、更为底层的 Rootkit，其基本思想是在操作系统加载之前就植入恶意代码，并以“老资格”的身份阻断杀毒软件对自己的检测和查杀。基于主动免疫的可信计算技术是对 Bootkit 的有效威胁。

第三区间

应用软件通信机制

第21章　通信框架

现代软件最大的特点就是通信，很难想象一个功能复杂但没有通信功能的软件怎样才能正常工作并为人们所认可。Windows 系统中具有丰富的通信机制，支持包括 TCP/IP、文件读写、串口、USB 等多种形态在内的 I/O 通信方法。

Windows 中为了支持 I/O 通信而设计了若干种通信模型。设计这些模型就是为了方便应用进程驾驭通信机制，并且使通信的效率尽可能提高。这里的 I/O 不仅仅指 TCP/IP 通信，也可以是文件读写或串口通信，因此可以看作广义的通信机制。但这里我们主要以狭义的 TCP/IP 通信为例讲述通信模型和框架。

随着通信模型和编程语言的发展，一些开源组织和商业组织又基于这些模型做了二次封装，形成了一系列“深加工”的通信模型产品，这些产品更适合某种语言、某种场景或某种数据，更便于应用进程使用，这就是通信框架。

本章首先讲述 Windows 系统的三类通信模型，让读者明白原生的通信模型是什么样子的；继而讲述“深加工”的通信框架，使读者清楚当前有哪些开源、易用的通信框架，它们都有什么特点。通信框架包含两个方面，一个是框架，一个是消息队列，但我们把它们都归类为软件栈中比通信模型更上层的通信框架。

21.1　Windows 系统通信模型

Windows 异步通信模型分为 3 类共计 6 种，分别是 I/O 完成端口模型、select 模型、WSAAsyncSelect 模型、WSAEventSelect 模型、基于事件对象方式的重叠 I/O 模型和基于完成例程方式的重叠 I/O 模型。这 6 种模型各有特点，其中 I/O 完成端口模型是 Windows 中大规模并发系统的首选通信模型，也是这些模型中通信效率最高、并发容量最高同时复杂度也最高的“三高”模型；而 select 模型、WSAAsyncSelect 模型、WSAEventSelect 模型可以作为 select 模型的三个子类；基于事件对象方式的重叠 I/O 模型和基于完成例程方式的重叠 I/O 模型又可作为重叠 I/O 模型的两个分类。

鉴于 I/O 完成端口模型的“三高”特性，我们首先来看完成端口模型的实现机制。

21.1.1　I/O 完成端口模型

I/O 完成端口（Input/Output Completion Port，IOCP）是 Windows 系统中最重要的 I/O 模型，也是性能最高的异步通信模型，它为 Windows 所独有。我们经常使用的 ACE（Adaptive Communication Environment，自适应通信环境）框架、Apache 框架等就是采用 I/O 完成端口实

现的,这种模型在处理海量连接的应用场景中优势非常明显。I/O 完成端口本身是一个 Windows 内核对象,对象类型为 IoCompletion,它是在 I/O 系统初始化时被创建的,本质上是一个内核队列,具有如下特点:

- 创建了一个线程池专门用于从内核队列中获取网络事件。
  - 线程池中线程的数量阈值与 CPU 核数相匹配,从而使线程切换开销降到最低。
  - 传统的 I/O 模型要么是一个 I/O 对应一个线程(one-thread-per-client),要么是所有 I/O 对应一个线程。前者在海量连接的情况下线程切换开销巨大;后者造成大部分线程同步等待,通信效率很低。I/O 完成端口模型正好折中了上述两种策略。
- 交给 I/O 线程的都是完成事件,应用进程无需采取其他步骤进一步处理。完成事件结构中已经携带了操作结果(例如已收到的数据、数据长度等),提高了异步性能和开发便捷性。也就是说,线程收到 I/O 完成通知的时候,操作结果都已经准备好了,根本不需要自己再去获取操作结果,服务非常周到。

Windows API 为 I/O 完成端口模型定义了若干接口函数:

- **CreateIoCompletionPort**:对应了 NtCreateIoCompletion 和 NtSetInformationFile 这两个系统调用。前者本质上是调用 KeInitializeQueue,作用是创建 I/O 完成端口,初始化线程池,通过参数指定池中线程数量,并创建和初始化 I/O 完成端口队列;后者的作用则是关联文件对象与刚刚创建的 I/O 完成端口,并注册文件对象的完成事件通知例程。
- **GetQueuedCompletionStatus**:对应了系统调用 NtRemoveIoCompletion(本质上是调用 KeRemoveQueue),作用是从 I/O 完成端口队列中移除一个 I/O 完成事件,释放 IRP,并通知上层应用线程取走 I/O 完成结果。
- **PostQueuedCompletionStatus**:对应了系统调用 NtSetIoCompletion(本质上是调用 KeInsertQueue),作用是主动构造一个 I/O 完成事件并插入 I/O 完成端口队列中,多用于结束当前 I/O 线程池中的线程。

I/O 完成端口的运行流程如图 21－1 所示,具体如下:

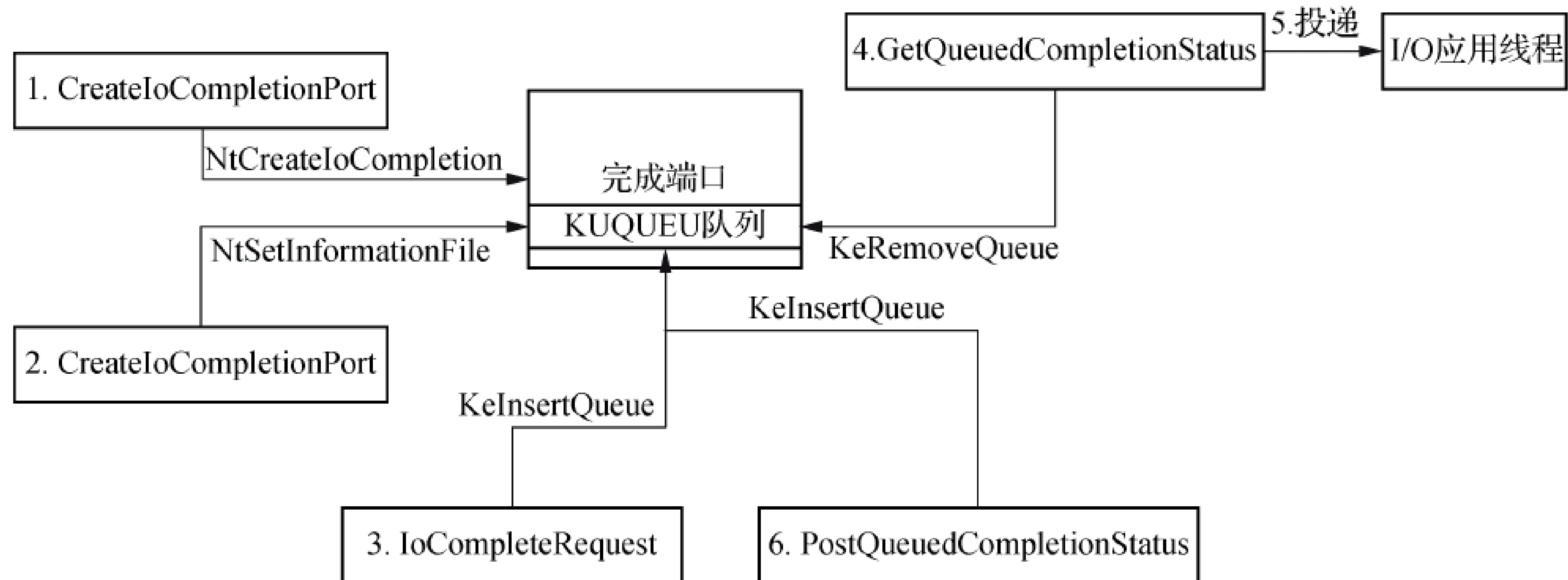


图 21－1　I/O 完成端口全生命周期流程

（1）调用 **CreateIoCompletionPort** 创建 I/O 完成端口。

（2）调用 **CreateIoCompletionPort** 关联 I/O 完成端口与文件对象（文件对象包括 socket、文件句柄等），使得 I/O 的文件对象纳入 I/O 完成端口的通知体系。

（3）I/O 驱动程序收到了 I/O 完成事件（这些事件包括写操作已完成、读操作已接收到数据并拷贝到了 IRP 的缓冲区、socket 的 Accept 事件已完成、网络五元组信息已拷贝到 IRP 的缓冲区等）后调用 IRP 的完成函数 IoCompleteRequest，将 IRP 的缓冲区信息转换为 I/O 完成事件并调用 KeInsertQueue 入队。

（4）I/O 完成端口线程池中的线程调用 **GetQueuedCompletionStatus**，从队列中获取 I/O 完成事件并投递给应用进程。应用进程中的 I/O 线程可能就是 I/O 完成端口线程池中的线程。

（5）当不再需要 I/O 完成端口时，调用 **PostQueuedCompletionStatus** 向队列中投递 I/O 完成事件（特殊标识，使线程能够知道这是结束线程的操作）。GetQueuedCompletionStatus 获得 I/O 完成事件后主动终结当前线程。当线程池中所有线程均终结退出后，线程池也就终结了。

从上述流程可以看出，I/O 完成端口模型本质上是个队列操作，入队（KeInsertQueue）和出队（KeRemoveQueue）操作贯穿始终。

在实际系统中，线程池中的线程数量也可以不完全与 CPU 核（逻辑核）数一致。实验表明：当线程数量 = CPU 核数 ×2 +2 时，线程切换开销最低。而线程池中也不是每时每刻都保持规定阈值数量的线程，例如某线程被阻塞，I/O 完成端口线程调度器会检测到阻塞并开启新线程继续调用 GetQueuedCompletionStatus，当阻塞线程被唤醒后，线程池中的线程数就比规定阈值多了 1 个。但这并没有什么大碍，下次其他线程被阻塞时线程调度器不再创建新线程就是了。因此线程数量阈值是个动态的建议值，并不是不能逾越的。

下面我们以 socket 通信中的 TCP 监听和接收数据为例来讲解 I/O 完成端口模型的使用，如图 21 -2 所示。

（1）开启服务端主线程，创建 I/O 完成端口并初始化，由 CreateIoCompletionStatus 实现。

（2）主线程初始化 socket，并调用 Listen 接口监听，完成 TCP 监听 socket 的初始化，并将创建的监听 socket 与 I/O 完成端口关联绑定，由 CreateIoCompletionPort 实现。

（3）在该 I/O 完成端口上投递 TCP 监听端口的 AcceptEx 请求。

- 所谓“投递”，其实就是调用 AcceptEx、WSARecv 等函数将请求与 socket 绑定。
- AcceptEx 是 Windows 专有的网络操作扩展函数（在 mswsock. dll 中实现），而不是 WinSock2 提供的，AcceptEx 与 WinSock2 的 Accept 函数相比具有更高的效率，因为在发出 AcceptEx 调用的时候 socket 已经建立好了，而 Accept 却需要线程自己创建 socket。Windows 提供的这个函数方便了我们使用重叠 I/O 机制。
- 重叠 I/O 机制是通过重叠结构 Overlapped 实现的，Windows 中的异步 I/O 通信都是通过这个数据结构实现的。

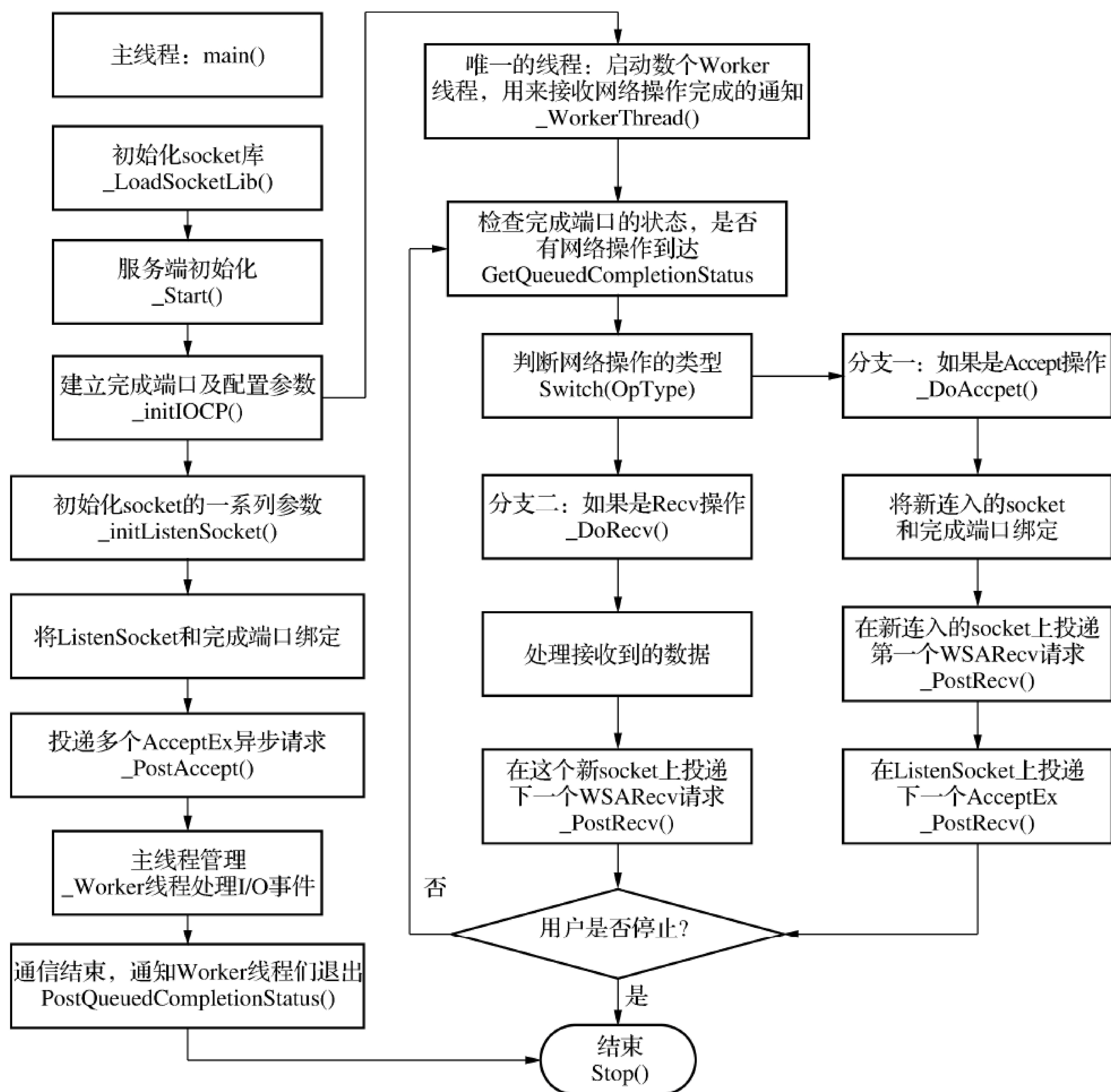


图 21－2　基于完成端口的网络监听与收包流程

➢ 所谓“重叠”是指执行 I/O 请求的时间与线程执行其他任务的时间是重叠的（overlapped），几乎所有 Windows 异步网络操作接口如 WSASend、WSARecv 等都以重叠结构为参数。重叠结构中携带了网络操作的标识，方便区分异步操作对应了哪个 socket 的哪一次请求。

➢ 重叠结构具体如下所示：

```
typedef struct _OVERLAPPED{
     DWORD Internal;
     DWORD InternalHigh;
     DWORD Offset;
     DWORD OffsetHigh;
     HANDLE hEvent;   //核心参数,这个就是连接异步 I/O 和应用程序的桥梁
}OVERLAPPED, * LPOVERLAPPED
```

（4）等待 TCP 客户端的连接请求。

（5）线程池中的线程调用 GetQueuedCompletionStatus 获取网络完成事件。当没有完成

事件时线程阻塞；当发生了网络完成事件时线程被唤醒，并判断具体是哪种事件：

➢ 若为 TCP 客户端的连接请求，则工作线程调用 Accept 函数响应该请求。新连入的 socket 作为通信 socket 与 I/O 完成端口关联，并在该通信 socket 上投递 TCP 报文接收的 WSARecv 请求，表明已经准备好接收数据了，同时也在 TCP 监听 socket 上再次投递一个 AcceptEx 请求（少一个补一个）。

➢ 若为 TCP 连接的 Recv 请求，则证明网络上有数据待读取。工作线程处理收到的数据并在通信 socket 上再投递一个 WSARecv 请求（少一个补一个）。

（6）若需要终止/完成端口，则通过 PostQueuedCompletionStatus 向线程池投递特殊标识消息，池中各线程通过 GetQueuedCompletionStatus 获取到特殊标识消息从而终结当前线程。

I/O 完成端口模型使用少量的线程处理大量的并发通信，这也要求 I/O 线程尽量不要有阻塞行为。因为线程池中的一个 I/O 线程往往对应了很多条并发通信链路，若某个线程阻塞，则该线程对应的所有链路必然阻塞，从而造成不可预计的后果。

21.1.2 select 模型

select 函数也叫多路分离函数，建立在这个分离函数基础之上的 I/O 模型就称为多路复用（select）模型。在 ACE 框架中，Windows 版本的反应堆（Reactor）模型就是采用 select 机制实现的（Linux 下采用 epoll 机制）。select 模型通过一个 fd_set 集合管理所有 socket，其本质上是一个轮询机制，一个 select 线程可以处理多个 socket 请求，工作过程如下：

（1）应用进程初始化 fd_set 集合并调用 select 函数。

（2）select 函数对 fd_set 集合中的所有 socket 轮询检查，如果某个 socket 状态有改变，则将有变化的 socket 记录下来。

（3）select 轮询完成，将记录的状态有变化的 socket 返回给应用进程，应用进程根据 socket 关联的上下文获取发生的具体事件和 I/O 数据。

从上述步骤我们发现，select 模型本质上是同步的，模型循环等待事件的发生，而不是像 I/O 完成端口模型那样“被通知”。前者主动，后者被动。

select 函数的原型如下所示：

```
int select(
  Int nfds,                         //传入 0 即可
  fd_set * readfds,                 //可读 socket 集合
  fd_set * writefds,                //可写 socket 集合
  fd_set * exceptfds,               //错误 socket 集合
  const struct timeval * timeout);  //select 函数等待时间
```

可见，select 模型将 fd_set 集合分为三类：读、写、错误（含带外数据），而 TCP 的 Accept 也被认为是读操作的一种而归入读集合里面。受限于 fd_set 集合的最大数量，select 模型最多支持 1 024 个并发的连接。fd_set 集合也是个很简单的数据结构，如下所示：

```
#ifndef FD_SETSIZE
#define FD_SETSIZE      64
#endif /* FD_SETSIZE */
```

```
typedef struct fd_set {
    u_int fd_count;                //fd_set 中 socket 的数量
    socket fd_array[FD_SETSIZE];   //socket 数组
}fd_set;
```

select 函数返回时，fd_set 中存放着状态发生改变的 socket，供应用进程处理。

从图 21 - 3 可以看出，select 模型是存在阻塞的（等待数据到达），但不是一个 socket 一个 socket 地等，而是一口气等待 fd_set 集合中所有的 socket。只有集合中所有 socket 的状态都没有改变时线程才阻塞。

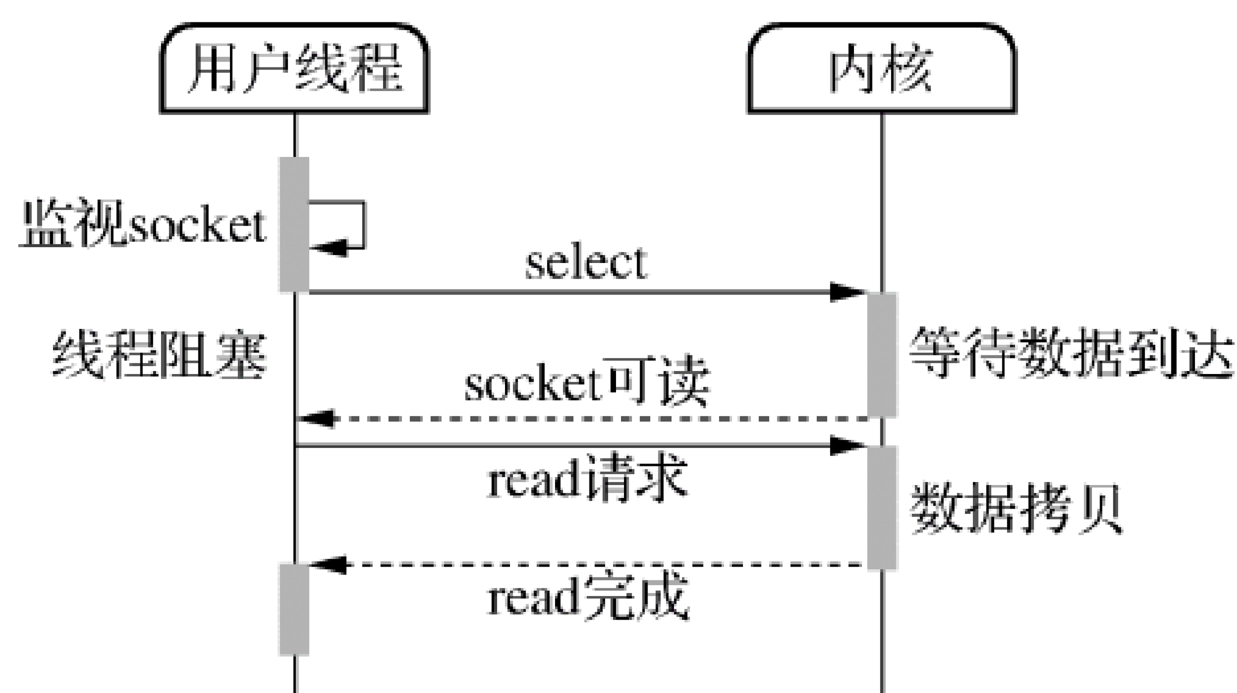


图 21 - 3　select 模型视图

fd_set 集合也可以只保存一个 socket，在一些特殊场景下，我们需要一个 socket 对应一个线程，这样可以随意阻塞 I/O 线程。这种情况下 select 模型就是个很好的选择，这也是它与 I/O 完成端口模型相比的特殊之处。但是 select 模型没有像 I/O 完成端口模型那样“服务到家”，侦测到状态改变后，还需要应用进程自己调用 accept 或 recv 接口来进一步处理收到的连接请求或数据。

当然 select 模型的不足之处也是很明显的：

- 每次 select 函数执行时都需要将 fd_set 从用户态空间拷贝到内核态空间，有结果需要返回时再将内核态中的 fd_set 拷贝到用户态的出参中。当这种动作非常频繁时，拷贝和切换的开销不可忽略。
- select 函数支持的 fd_set 集合数最大只有 1 024，即最多只能轮询 1 024 个 socket，这个数量远远小于 I/O 完成端口支持的 socket 数。fd_set 集合数的原始定义只有 64，可以通过重定义 FD_SETSIZE 宏来扩展最大集合数，但最大也只能扩展到 1 024。
- fd_set 集合数比较大时，select 函数的轮询开销不可忽略。

除了 select 模型，Windows 还支持 WSAEventSelect 和 WSAAsyncSelect 两种“类 select 模型”，前者也叫“事件选择模型”，后者则被称为“异步选择模型”。

1）事件选择模型 WSAEventSelect

事件选择模型基于事件通知，根据事件是否被触发来进行相应处理。这些事件类型包括：

- **FD_READ**：socket 可读通知；
- **FD_WRITE**：socket 可写通知；
- **FD_OOB**：带外数据到达通知；
- **FD_ACCEPT**：TCP 服务端已接受了本地客户端的连接请求；
- **FD_CONNECT**：TCP 客户端向本地服务端发来了连接请求；
- **FD_CLOSE**：socket 关闭通知；
- **FD_QOS**：QOS 变化的通知；
- **FD_GROUP_QOS**：组 QOS 变化的通知；

- **FD_ROUTING_INTERFACE_CHANGE**:与路由器的接口发生变化的通知;
- **FD_ADDRESS_LIST_CHANGE**:本地地址列表发生变化的通知;
- **FD_ALL_EVENTS**:代表所有事件类型。

事件选择模型的事件关联注册函数是 WSAEventSelect,事件重置函数为 WSAResetEvent,事件分离函数是 WSAWaitForMultipleEvents,事件关闭函数则为 WSACloseEvent。事件选择模型本质上还是轮询等待,也受限于 socket 的最大处理数量(64 个),且数量无法更改。

事件选择模型的运行流程如下:

(1) 创建用于存放所有事件对象的数组。

(2) 使用 **WSACreateEvent** 创建一个事件对象(类型如上),并将事件对象添加到数组。

(3) 使用 **WSAEventSelect** 将 socket 与上述事件对象关联起来。

(4) 调用 **WSAWaitForMultipleEvents** 等待网络事件发生。

(5) 事件发生后使用 **WSAEnumNetworkEvents** 查询具体发生了哪种类型的事件。

(6)进一步处理事件(如接受连接、获取数据等),并重复执行步骤(4)和(5)。

2) 异步选择模型 WSAAsyncSelect

异步选择模型与事件选择模型很相似,后者在有事件发生时激活一个事件对象,而前者则是发送一个消息给指定的窗口,应用进程在窗口响应例程中处理这些消息。

异步选择模型支持的网络事件类型与事件选择模型定义的事件类型完全一致。不过,由于在异步选择模型中消息是投递到窗口上的,因此要求应用进程必须具有窗口。WSAAsyncSelect 本质上是利用 Windows 窗口消息队列机制来通知我们设定的窗口过程例程。

21.1.3 重叠 I/O 模型

重叠 I/O 模型使用重叠数据结构(WSAOverlapped)来投递 I/O 操作,可以一次投递多个 I/O 操作,并具有很高的读写(收发)并发效率。与 select 模型相比,重叠 I/O 模型可以将读写结果直接拷贝到用户进程的缓冲区而不需要另外调用接口来进一步接收(RECV)或接受(ACCEPT)。也就是说,当重叠 I/O 模型收到读操作时,其收到的结果数据已经存在于用户进程缓冲区了。

I/O 完成端口模型中已经对重叠结构做了描述,这里要强调的是存在两种方法管理重叠 I/O 的完成情况:事件对象方式和完成例程方式。其实 I/O 完成端口也是基于重叠结构的,但一般将 I/O 完成端口单独作为一类模型。

1. 事件对象方式

事件对象方式要求将 I/O 事件与重叠结构关联在一起,同时摒弃 WinSock2 库中的 send、sendto、recv 等接口转而使用 Windows 原生的 WSARecv、WSASend 等接口,因为只有后者才能将重叠结构作为参数传递进内核。

WSARecv 函数原型如下所示,其倒数第二个参数即为重叠结构:

```
int WSArecv(
    SOCKET s,                        //投递这个操作的 socket
    LPWSABUF lpBuffer,               //接收缓冲区,与 Recv 函数不同
    LPDWORD lpNumberOfBytesRecvd,//如果接收操作立即完成,这里会返回函数调用
    LPDWORD lpFlags,                 //默认为 0
    LPWSAOVERLAPPER lpOverlapper,//绑定的重叠结构
    LPWSAOVERLAPPER_COMPLETION_ROUTINE lpCompletionRoutine   //一个回调
);
```

重叠 I/O 模型采用 **WSAWaitForMultipleEvents** 方法来等待需要的激活事件,这个函数最多只支持 64 个事件,也就是说一个重叠 I/O 模型最多支持 64 个 socket。当然可以采用多线程的方式来解决上述限制。重叠结构中隐含着 I/O 结果,因此还需要 **WSAGetOverlappedResult** 方法来查询操作结果。

我们仍以 TCP 服务端的连接请求接受和数据接收为例来讲述基于事件对象方式的步骤:

(1) 创建 socket 并进行服务端监听(监听 socket)。

(2) 接受客户端的连接请求。

(3) 为接受的 socket(通信 socket)创建重叠结构,并为之分配事件句柄(关联 I/O 事件)。

(4) 在通信 socket 上投递 **WSARecv** 操作,并将重叠结构作为参数传递进去。

(5) 调用 **WSAWaitForMultipleEvents** 函数等待关联的 I/O 事件被激活。

(6) 若有 I/O 事件激活,则通过 **WSAGetOverlappedResult** 函数获取完成的重叠结构,以便将 I/O 结果传递给应用层。

(7) 调用 **WSAResetEvent** 函数将事件重置。

(8) 在通信 socket 上再次投递 WSARecv 操作。

(9) 重复执行步骤(5)~(8)。

2. 完成例程方式

所谓完成例程,简单来说就是回调函数。我们设置事件处理回调函数到模型中,当要监控的 I/O 事件发生时,回调函数会被模型调用,只需要实现回调函数中的处理逻辑即可。完成例程方式与事件对象方式在性能上差不多,但前者不受制于 64 个等待事件的限制。

回调函数的原型如下所示:

```
Void CALLBACK_CompletionRoutine(
    DWORD dwError,                    //重叠操作的状态
    DWORD cbTransferred,              //实际传输的字节量
    LPWSAOVERLAPPED lpOverlapped,     //重叠结构
    DWORD dwFlags                     //返回操作结束时可能用的标志,不常用
);
```

也就是说,我们要提前设置这样一个回调函数到模型中。在前文介绍事件对象方式的时候,WSARecv 函数的最后一个参数就是一个函数指针,这其实就是完成例程。Windows 在设计这些接口的时候就将重叠 I/O 的两种情况都考虑到了。

基于回调函数的方法和基于事件对象的方法在使用步骤上基本一致,只是前者在投递

WSARecv 的时候要将回调函数设置进去,并且不需要设置事件对象。

完成例程方式的原理是 APC(异步过程调用)机制。我们在前文中有过描述,应用进程(线程)设置用户态的回调指针,借用 APC 机制保存到 APC 数据结构里,APC 结构通过队列方式被编排到每个 KTHREAD 中,在线程切换时这些 APC 会得到执行,也就是回调函数会被执行。

在 I/O 驱动程序中,每次 IRP 完成,都会通过 IoCompleteRequest 函数将本次 I/O 操作的结果挂到对应线程 KTHREAD 的 APC 队列中。一个生产者,一个消费者,如此构成了完成例程的基本循环机制。

21.2 通信框架

通信框架是对通信模型的二次封装,目的是便于不同的语言使用,并且降低使用门槛,方便应用程序开发。至于通信框架是否就比通信模型更易上手、门槛更低,这也是多年来业界一直有争论的地方。当然,仁者见仁智者见智,无论怎样通信框架已经深深融入我们的开发中了,因为总体来讲通信框架确实要比通信模型更友好、更适合大规模应用。

目前市面上的通信框架百家争鸣,有针对单节点通信的框架,也有基于分布式的框架;有针对 C/C++ 语言的,也有针对 Java 或脚本语言的;有开源的,也有闭源的;有社区贡献共享的,也有商用开发和付费使用的;有基于 TCP/IP 协议的,也有针对非 TCP/IP 协议的;有针对专门应用场景的(例如视频会议),但更多的是基于通用场景的。但无论怎样,绝大多数通信框架都遵循某种事件多路分离模型,例如 IOCP 模型、epoll 模型、select 模型等,在此基础上实现了通信的事件通知和数据读写处理。

21.2.1 ACE 框架

ACE(Adaptive Communication Environment,自适应通信环境)框架是 Windows 下处理高并发的首选框架。但 ACE 也是一个高复杂度和比较臃肿的重量级通用框架,因为它不仅仅包含了 I/O 通信相关的组件,还包括了并发与同步、内存管理、定时器管理、信号通知、线程管理、文件管理、软件配置管理、容器管理等多种组件,并且兼容 Windows 和 Linux 两套操作系统(支持 POSIX 语义的操作)。

ACE 框架是由 C++ 语言开发的跨平台开源框架,由操作系统适配层、C++ 封装适配层、框架和网络服务层构成,如图 21 – 4 所示。

➢ **操作系统适配层**:提供操作系统无关性支持,ACE 框架的使用者只要很少的工作量就可以切换操作系统平台。操作系统适配是由框架的 ACE_OS 类实现的。

➢ **C++ 封装适配层**:包括一些 C++ 包装类,用于构建可移植且类型安全的 C++ 应用和事件多路分离机制等,其中包括:

- 并发与同步类——对操作系统多线程 API 进行了抽象封装;

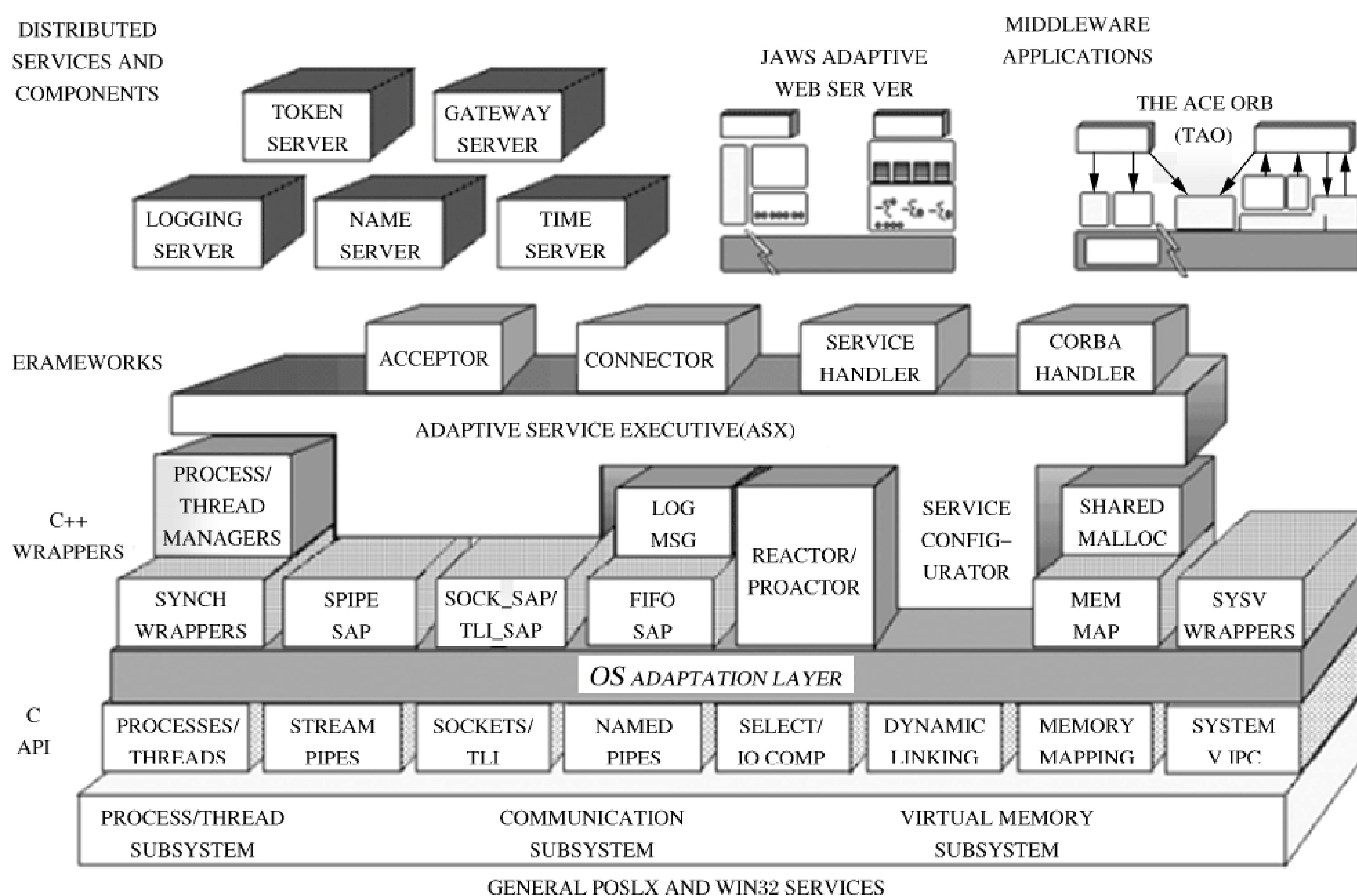


图 21－4　ACE 框架的层次(图片来自 ACE 官网)

- IPC 类——支持 socket 接口、命名管道、消息队列等;
- 内存管理类——支持内存分配和释放管理;
- 定时器类——支持定时器调度和取消,并且也封装了高分辨率的定时器;
- 容器类——支持 STL 风格容器,如 Map、List 等;
- 线程管理类——支持创建和终止线程,支持线程局部存储机制。

➢ **框架和网络服务层**:这是基于操作系统的通信模型构筑和封装的一层,包括:

- 事件处理组件——支持基于 I/O 通信、定时器、信号和同步等事件的处理器;
- 网络连接和服务初始化组件——支持连接器、接受器组件,用于 TCP 握手协商,这样规划便于握手和连接通信的去耦合;
- 服务配置组件(ACE Service Configurator)——对于安装或运行时配置的软件支持动态启动、挂起和配置;
- 流组件(ACE Stream)——用于分层软件栈的开发,支持针对软件栈中每一层中流经的数据进行修改和传递。

ACE 框架实在是太庞大了,限于篇幅和主线,本章只介绍 ACE 框架的 I/O 通信部分。

操作系统的 I/O 通信可分为阻塞、非阻塞同步、非阻塞异步三种方式,相较而言,非阻塞异步方式的性能和扩展性最好,而目前大部分通信场景也都是支持这种方式的,ACE 框架的通信组件就是非阻塞异步 I/O 组件。

非阻塞异步方式的 I/O 通信最重要的部件就是事件多路分离器(Event Demultiplexer)。所谓事件多路分离就是将来自于多个事件源(socket)的 I/O 事件分离出来并派发到对应的





事件处理器(Event Handler,即应用进程中的 I/O 线程执行的函数)。

ACE 框架中有两个基础设施负责多路分离:前摄器(Proactor)和反应堆(Reactor)。前者基于 IOCP,多用于 Windows 下的 I/O 高并发场景;后者基于 epoll,多见于 Linux 下的高并发场景。但实际上在 Linux 下也有基于 epoll 模型实现的前摄器,但其效率不会超过反应堆,因此这里我们只介绍基于 IOCP 的前摄器和基于 epoll 的反应堆。

1. 前摄器

前摄器是 ACE 框架中的异步网络处理器,其本质是完成事件 + 多路分离,其依赖的模型是 IOCP。前摄器的运行框架如图 21－5 所示。

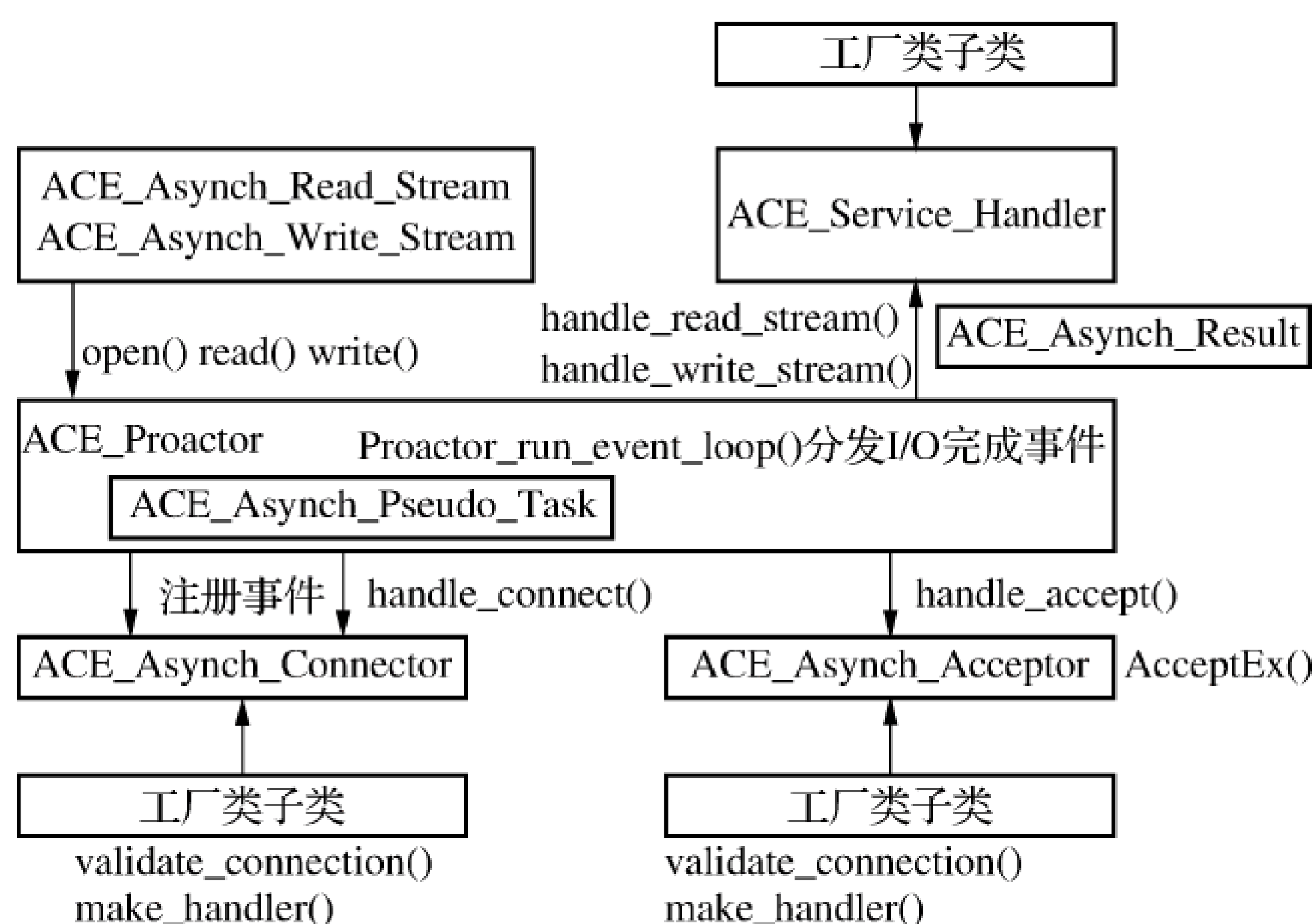


图 21－5　前摄器运行框架

前摄器运行框架由以下几个组件构成(仅以 TCP 通信方式为例):

- **ACE_Asynch_Connector**:连接器,用于 TCP 客户端向服务端发起连接请求。这是一个模板类,指定了它的派生类的服务处理器 ACE_Service_Handler 的派生类类型。
- **ACE_Asynch_Acceptor**:接受器,用于 TCP 服务端接受客户端发起的连接请求。这也是一个模板类,指定了它的派生类的服务处理器 ACE_Service_Handler 的派生类类型。
- **ACE_Asynch_Read_Stream**:用于发起异步读操作。
- **ACE_Asynch_Write_Steam**:用于发起异步写操作。
- **ACE_Service_Handler**:服务处理器,包含了一些回调函数,当 I/O 操作完成后会自动触发对应的回调函数。这是一个工厂类,可以基于该工厂类实现自己的子类。
- **ACE_Asynch_Result**:表示针对每个异步操作的操作结果。
- **ACE_Proactor**:ACE 前摄器,用于循环执行多路分离,包括在 Windows 下基于 I/O 完成端口模型的 ACE_WIN32_Proactor 类,以及在 POSIX 系统上的 ACE_POSIX_Proactor 类,例如 Linux 中基于 epoll 模型的前摄器。

ACE_Asynch_Connector 除了提供 open(与前摄器绑定)、connect(向 TCP 服务端发起建

立连接请求)、cancel(关闭连接)等这些基本方法外,还提供了三个主要方法,它们也都是连接时被前摄器回调的方法,如下所示:

```
make_handler(void);
//创建一个新的服务处理器 ACE_Service_Handler,其派生类必须重新实现这个方法
validate_connection (const ACE_Asynch_Connect::Result& result,
                     const ACE_INET_Addr &remote,
                     const ACE_INET_Addr& local)
//连接即将建立成功时前摄器调用该函数,其人参带有 TCP 的客户端和服务端的四元组信息(客户端地址、客户
   //端端口号、服务端地址、服务端端口号),其派生类必须重新实现这个方法
handle_connect(const ACE_Asynch_Connect::Result &result)
//连接建立成功后调用该方法
```

ACE_Asynch_Acceptor 与 ACE_Asynch_Connector 大致相同,除了提供 open(与前摄器绑定)、accept(接受 TCP 客户端发来的建立连接的请求)、cancel(关闭连接)等基本方法外,也定义了三个连接时被前摄器回调的方法,其派生类也必须重新实现这三个模板方法,如下所示:

```
make_handler(void)
//创建一个新的服务处理器 ACE_Service_Handler
validate_connection(const ACE_Asynch_Accept::Result& result,
                    const ACE_INET_Addr &remote,
                    const ACE_INET_Addr &local)
//连接即将建立成功时用来回调 TCP 四元组
handle_accept(const ACE_Asynch_Accept::Result &result)
//连接建立成功后调用该方法
```

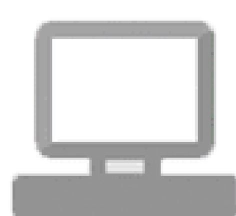
ACE 框架中的连接器和接受器二者本身就可以组成一个框架——Connect-Accept 框架,而前摄器被纳入了这个框架后更加方便了通信开发。

ACE_Asynch_Read_Stream 是前摄器的读操作类,提供了如下基本方法:

```
open(const ACE_Handler::Proxy_Ptr &handler_proxy,
     ACE_HANDLE handle,
     const void * completion_key,
     ACE_Proactor * proactor)
//将服务处理器、socket 句柄和前摄器关联起来
read(ACE_Message_Block &message_block,
     size_t bytes_to_read,
     unsigned long offset =0,
     unsigned long offset_high =0,
     const void * act =0,
     int priority =0,
     int signal_number =ACE_SIGRTMIN)
//发起异步读操作,注意,这里仅仅是发起操作而不是读到结果
```

ACE_Asynch_Write_Stream 是前摄器的写操作类,提供了如下基本方法:

```
open(ACE_Handler &handler,
     ACE_HANDLE handle =ACE_INVALID_HANDLE,
     const void * completion_key =0,
     ACE_Proactor * proactor =0)
//将服务处理器、socket 句柄和前摄器关联起来
write(ACE_Message_Block &message_block,
      size_t bytes_to_write,
      const void * act =0,
      int priority =0,
      int signal_number =ACE_SIGRTMIN)
//发起异步写操作,这里也仅仅是发起操作,而写的结果要通过服务处理器的回调方法才能获知
```





ACE_Service_Handler 是服务处理器基类,其派生类至少要实现以下三个方法:

```
open(ACE_HANDLE new_handle,
     ACE_Message_Block &message_block)
//连接器或接受器的 make_handler 方法被调用完后,该服务处理器派生类的 open 方法会被调用,在该方法
//中可以初始化读/写操作类等
handle_read_stream(const ACE_Asynch_Read_Stream::Result &result)
//异步读结果回调函数,其参数就是读操作完成后的结果,例如数据指针、数据长度等
handle_write_stream(const ACE_Asynch_Write_Stream::Result &result)
//异步写结果回调函数,其参数也是操作完成后的结果
```

ACE_Asynch_Result 本质上代表了一个重叠结构,我们在服务处理器类的异步读写回调函数中的参数都是 ACE_Asynch_Result 类型的。

ACE_Proactor 是 ACE 前摄器的基类,提供如下主要方法:

```
proactor_run_event_loop(PROACTOR_EVENT_HOOK =0)
//开启前摄器事件分离线程,针对而言,就是不断循环调用 GetQueuedCompletionStatus 来获取重叠结构
//并转换成 ACE_Asynch_Result 派生类型(例如 ACE_WIN32_Asynch_Result),投递给服务处理器的结果
//回调函数
proactor_end_event_loop(void)
//结束前摄器事件分离线程
```

前摄器运行框架下 TCP 客户端与服务端的协作如图 21-6 和图 21-7 所示。

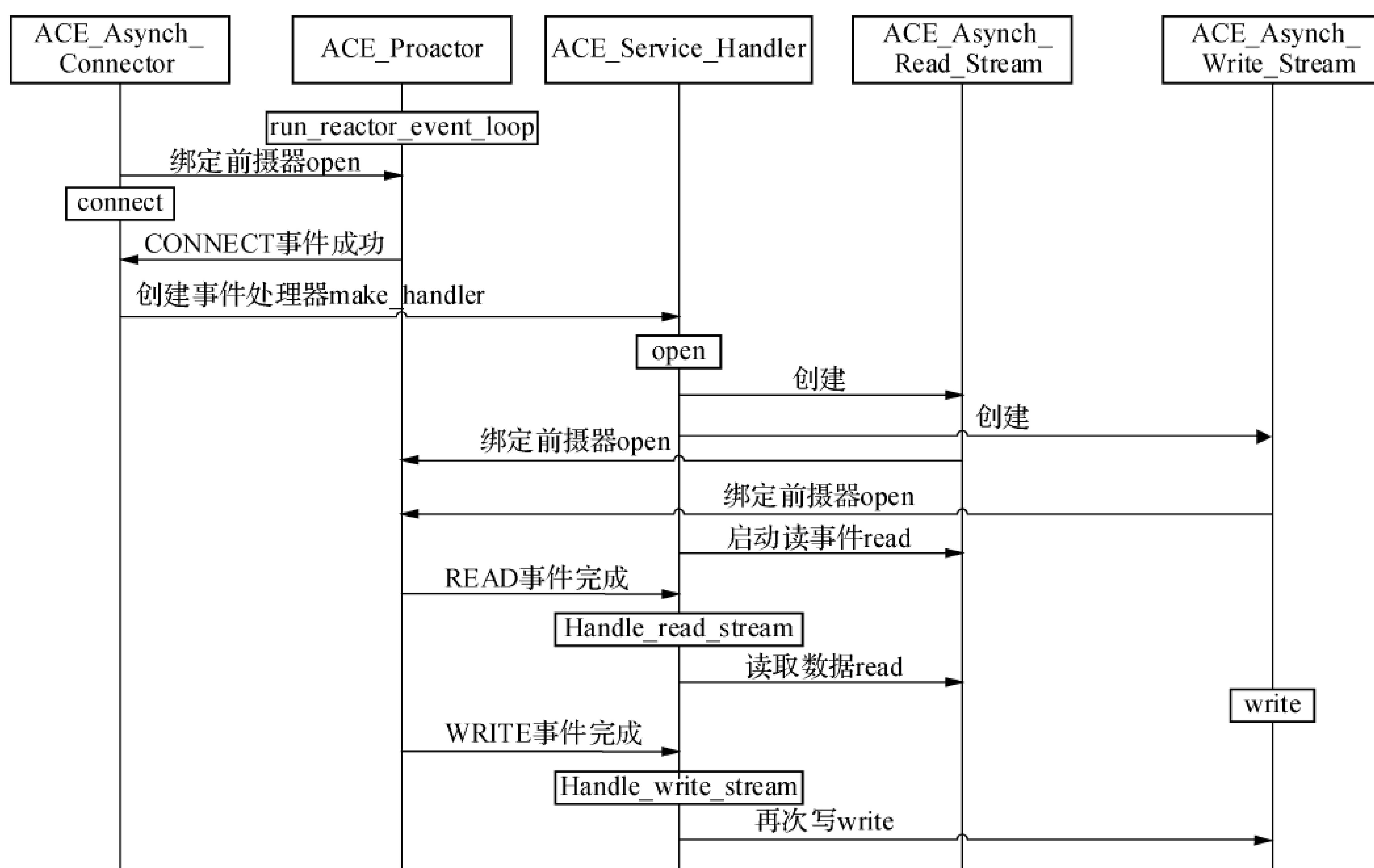


图 21-6　前摄器运行框架下 TCP 客户端的协作

2. 反应堆

相较于基于 IOCP 模型的前摄器,基于 epoll/select 模型的反应堆要简单不少。反应堆的运行框架如图 21-8 所示。

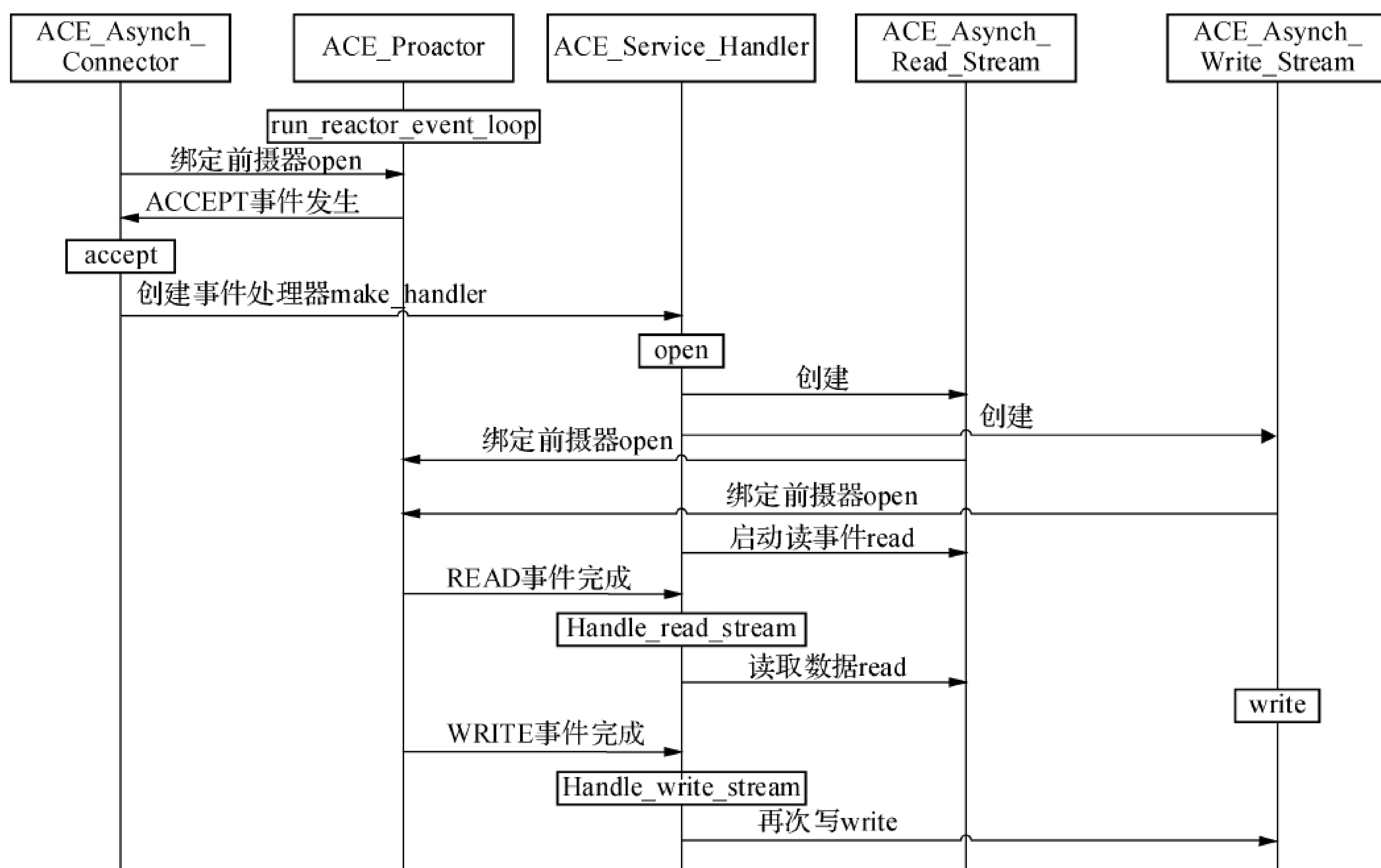


图 21－7　前摄器运行框架下 TCP 服务端的协作

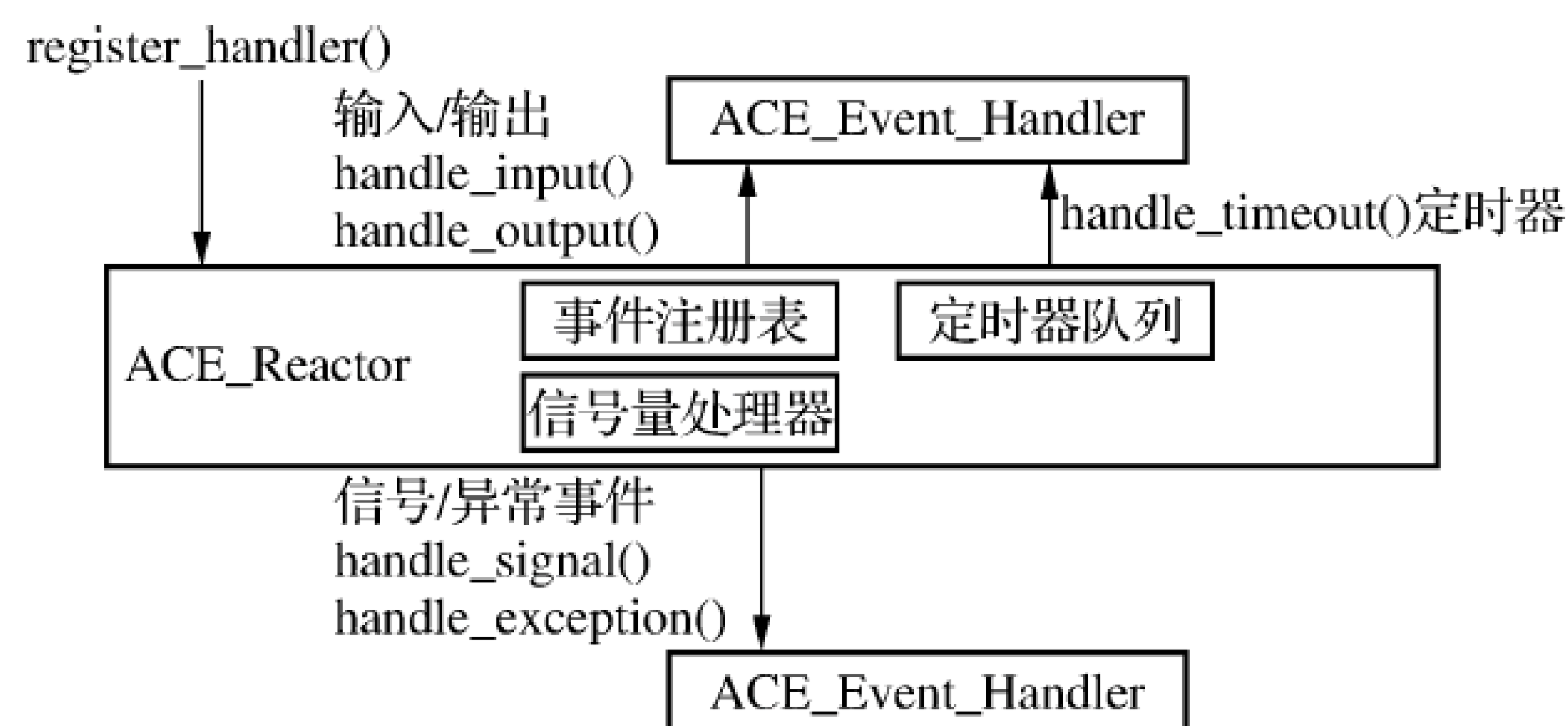


图 21－8　反应堆运行框架

我们仍以 TCP 为例来介绍反应堆运行框架。反应堆运行框架本身由两个组件构成：ACE_Event_Handler 和 ACE_Reactor。但是要实现 TCP 通信，还需要能实现 Connect-Accept 框架的 ACE_Connector 和 ACE_Acceptor 两个组件的支持。

➢ **ACE_Connector**：ACE 连接器，用于 TCP 客户端向服务端发起连接请求，可以类比前摄器运行框架中的 ACE_Asynch_Connector。ACE_Connector 是个模板类，指定了它的派生类事件处理器 ACE_Event_Handler 的派生类类型。ACE_Connector 除了提供 open、close、reactor、connect 等方法之外，还提供了以下三个重要方法：

```
make_svc_handler(SVC_HANDLER * &sh)
//创建一个新的事件处理器 ACE_Event_Handler 的派生类
connect_svc_handler(SVC_HANDLER * &svc_handler,
                    const ACE_PEER_CONNECTOR_ADDR &remote_addr,
```

```
                              ACE_Time_Value * timeout,
                              const ACE_PEER_CONNECTOR_ADDR &local_addr,
                              int reuse_addr,
                              int flags,
                              int perms)
//获取 TCP 四元组
activate_svc_handler(SVC_HANDLER * svc_handler)
//激活新创建的事件处理器,使 ACE_Event_Handler 的派生类调用自己的 open 方法
```

➢ **ACE_Acceptor**:ACE 接受器,用于 TCP 服务端接受客户端发起的连接请求,可以类比前摄器运行框架中的 ACE_Asynch_Acceptor。ACE_Acceptor 也是个模板类,指定了它的派生类服务处理器 ACE_Event_Handler 的派生类类型。ACE_Acceptor 除了提供 open、close 等方法外,还提供了以下三个重要方法:

```
make_svc_handler(SVC_HANDLER * &sh)
//创建一个新的事件处理器 ACE_Event_Handler 的派生类
accept_svc_handler(SVC_HANDLER * svc_handler)
//将参数传递给事件处理器
activate_svc_handler(SVC_HANDLER * svc_handler)
//激活新创建的事件处理器,使 ACE_Event_Handler 的派生类调用自己的 open 方法
```

➢ **ACE_Event_Handler**:ACE 事件处理器基类,可以类比前摄器运行框架中的 ACE_Service_Handler,TCP 的输入/输出均由该事件处理类的派生类负责。使用事件处理器之前必须要调用反应堆的 register_handler 方法将其注册到反应堆上,一般可在 open 方法中进行注册。除了 open、close 等方法外,ACE_Event_Handler 还至少提供了以下几个重要方法:

```
handle_input(ACE_HANDLE fd = ACE_INVALID_HANDLE)
//有数据可读,此时可以调用 recv 方法进行读数据的获取
handle_output(ACE_HANDLE fd = ACE_INVALID_HANDLE)
//上一次数据发送已完成,此时可以调用 send 方法继续发送数据
handle_close(ACE_HANDLE handle,ACE_Reactor_Mask close_mask)
//当事件处理器基类调用了 close 方法的时候,如果关闭已成功,则调用该方法通知事件处理器
handle_timeout(const ACE_Time_Value &time,const void * )
//定时器处理函数,当定时器到达约定时间时该方法会被调用
handle_exception(ACE_HANDLE fd = ACE_INVALID_HANDLE)
//异常事件的处理函数,包括带外数据的到达也是通过该方法获知(SIGURG 信号)
handle_signal(int signum,siginfo_t * =0,ucontext_t * =0)
//信号事件的处理函数
handle_exit(ACE_Process * )
//进程退出时被调用
```

➢ **ACE_Reactor**:ACE 反应堆基类,其具体实现包括 ACE_Select_Reactor、ACE_WFMO_Reactor、ACE_Dev_Poll_Reactor 等子类。

- **ACE_Select_Reactor**:无论是在 Windows 下还是在 Linux 下均使用 select 模型,支持的最大事件数为 64,但可通过修改配置扩展到 1 024。ACE_Select_Reactor 采用的是水平触发(条件触发)方式,只要满足条件就一直不厌其烦地通知你。
- **ACE_WFMO_Reactor**:由于这种反应堆使用了 WaitForMultipleObjectsEx 模型,而这是 Windows 下才有的模型,因此其只能运行于 Windows 下。ACE_WFMO_Reactor 采用的是边缘触发方式,每当状态变化时触发一个通知事件。在读数据过程中的所谓状态变化,是以读缓冲区中还有没有待读出的数据为判断基准的,而

诸如待读数据量从 1 000 减到 100 这样的变化是不算作状态变化的,只有从无到有或从有到无才会触发通知。

- **ACE_Dev_Poll_Reactor**:默认使用 poll 模型,但也可以通过修改配置更改为使用 epoll 模型。前者采用水平触发方式,而后者既可以采用水平触发方式也可以采用边缘触发方式。

无论哪种反应堆,都要通过 run_reactor_event_loop 方法执行事件多路分离循环,这一般是由专门的线程完成的。反应堆运行框架下 TCP 客户端与服务端的协作如图 21－9 和 21－10 所示。

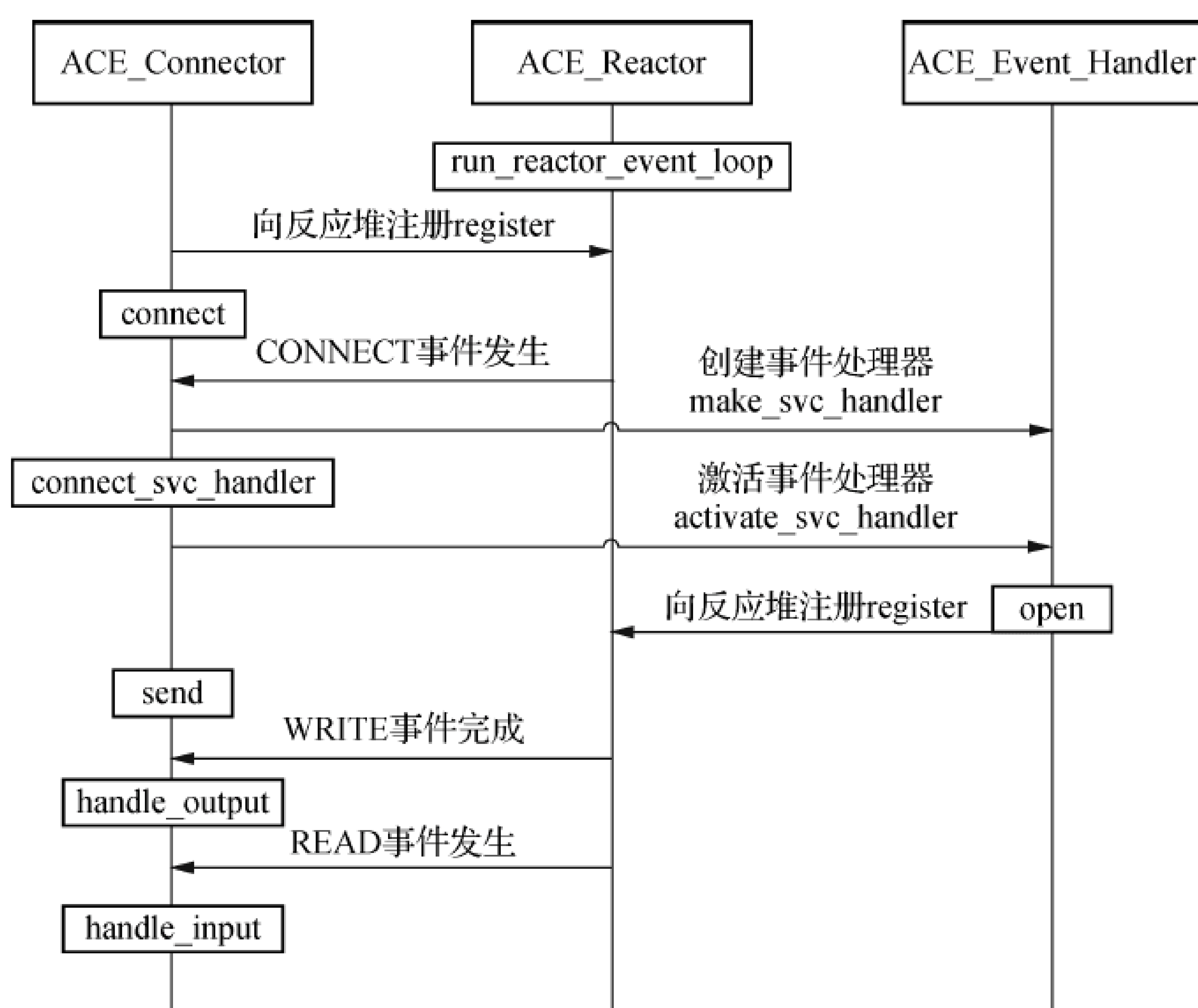


图 21－9　反应堆运行框架下 TCP 客户端的协作

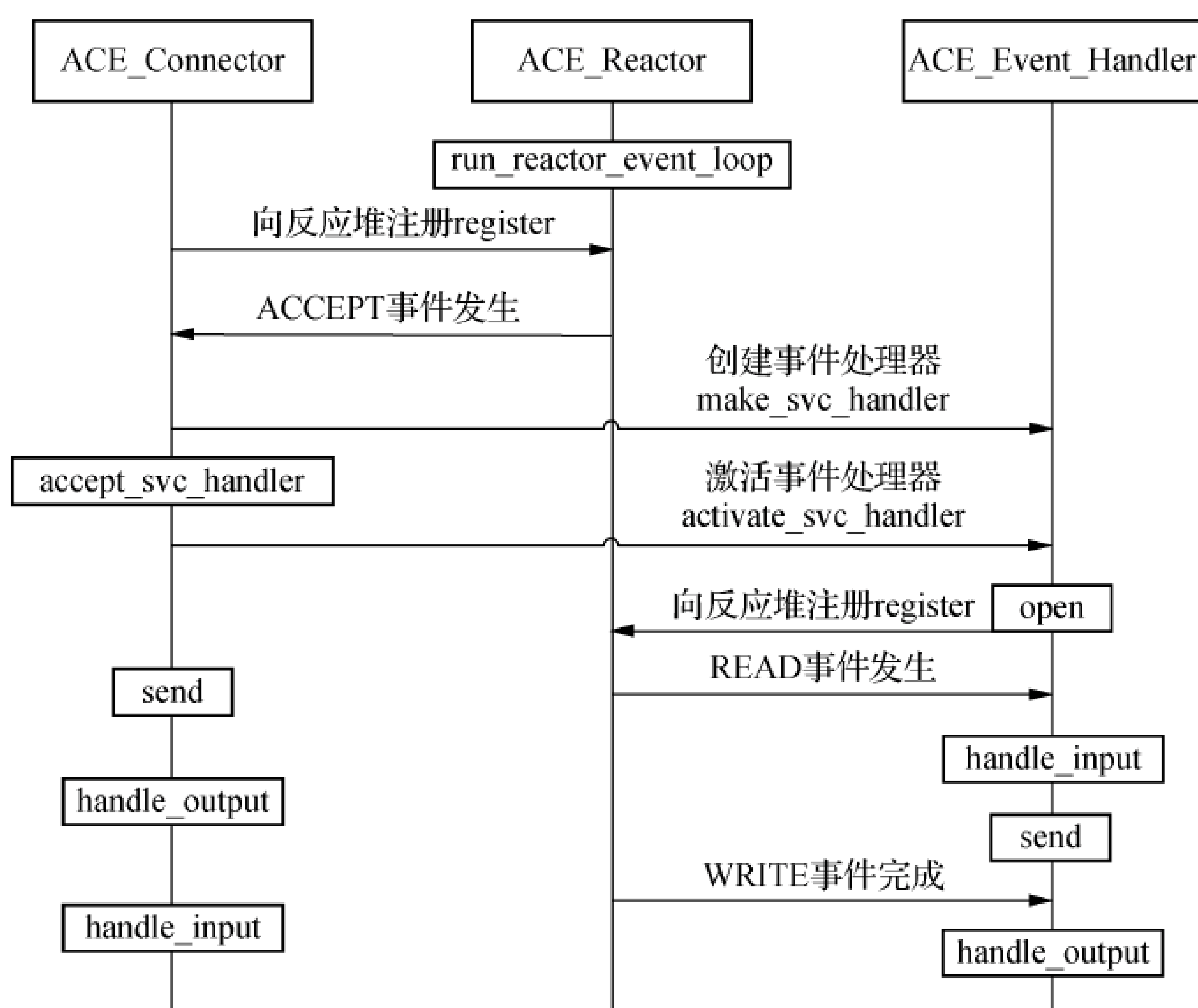


图 21－10　反应堆运行框架下 TCP 服务端的协作

在反应堆运行框架下，无论是客户端还是服务端，通信时都要创建一个事件处理器，并且两端的事件处理器都要向反应堆注册。

21.2.2 WebRTC 框架

WebRTC(Web Real-Time Communication，Web 实时通信)是一种基于浏览器页面的语音对话或视频通话的通信框架。因此，WebRTC 是一套针对细分场景的通信框架，其目的就是无插件化地实现视频会议、语音对讲、在线聊天等实时通信类功能。市面上已有不少视频会议产品采用了 WebRTC 框架。

WebRTC 的总体架构如图 21－11 所示。整个框架具有良好的 Web 接口，并同时支持 C++ 接口(Web 接口位于 C++ 接口之上)。支持 WebRTC 框架的主要有三大组件：音频引擎组件、视频引擎组件和传输引擎组件。其中，音频和视频引擎组件分别包括了视频采集和编解码部分，以及相应的防抖动缓冲等外围机制；传输引擎组件支持 P2P 方式，也支持封装和解封装，还支持对 RTP 的加密(SRTP)。

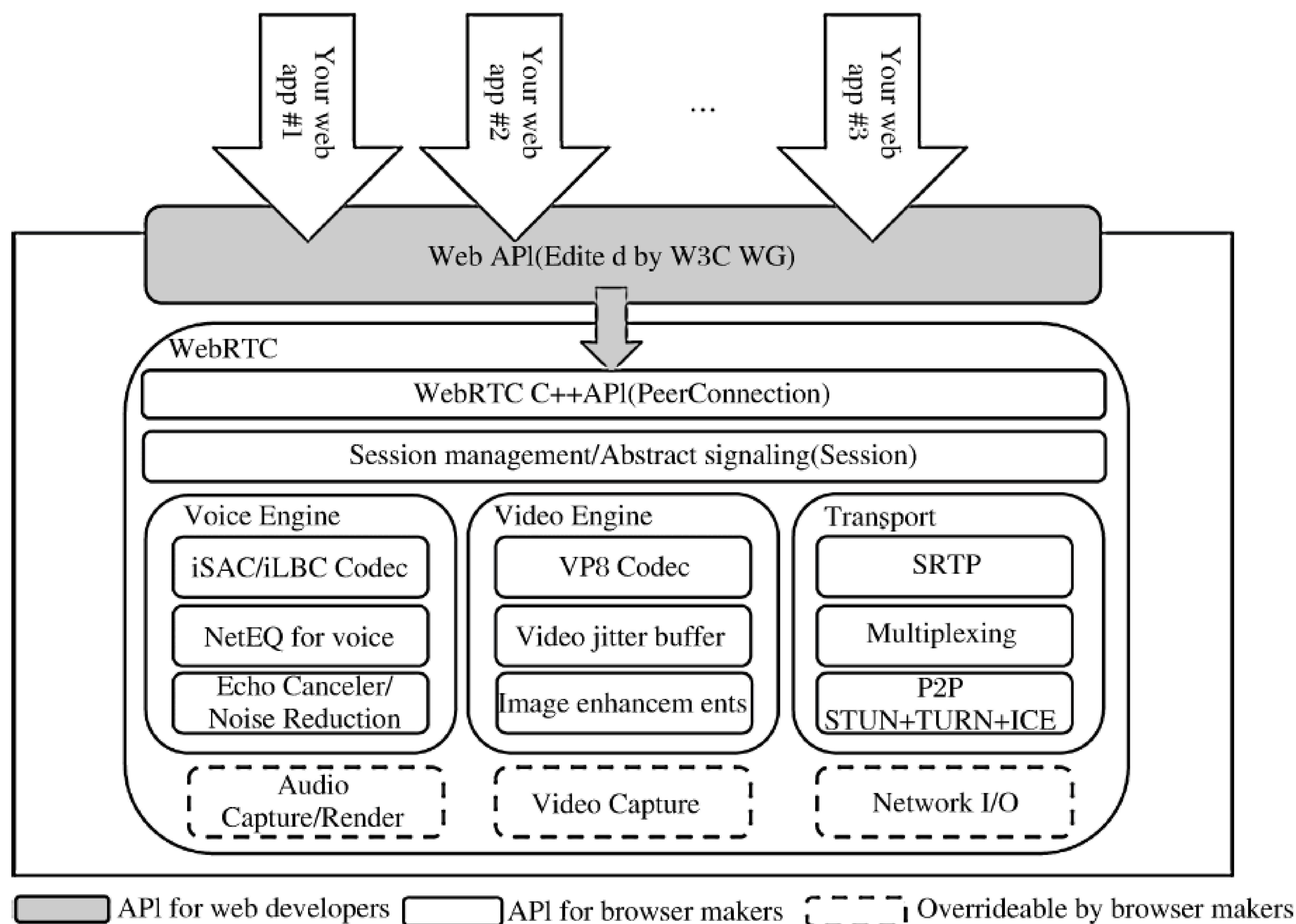


图 21－11　WebRTC 总体框架(图片来自 WebRTC 官网)

1. WebRTC 框架的特性

WebRTC 框架具有以下特性：

- **跨平台特性**：WebRTC 框架支持跨平台方案，兼容包括 Windows、Linux、Mac OS 和 Android 等在内的多套操作系统。在 PC、手机、Pad 等载体上均可以基于浏览器运行 WebRTC＋HTML5 代码。特别是在移动互联网时代，兼容 Pad、手机等载体尤为重要。

➢ **流媒体安全特性**：WebRTC 框架采用 SRTP/SRTCP 协议代替原来的 RTP/RTCP 协议，从流媒体封装上加强安全机制。同时，它也支持基于 HTTPS 方式的网页信息加密和基于 TLS 的四层协议加密。

➢ **私网穿透特性**：WebRTC 框架既可采用 STUN(Simple Traversal of UDP Though NAT，NAT 的 UDP 简单穿透)方式穿透 NAT(Network Address Translate，网络地址转换)，也支持 TURN(Traversal Using Relay NAT，通过 Relay 方式穿透 NAT)方式穿透 NAT，可以说私网穿透性是 WebRTC 框架最重要的特性。

➢ **网络自适应特性**：基于 SRTCP/RTCP 的反馈机制，可以动态调整流媒体发送侧码流的发送速度和发送带宽。

➢ **视频编解码特性**：WebRTC 作为由 Google 开源的流媒体框架，首先支持了 VP8 和 VP9 两种编解码算法(专利免费)。近年来由于 Google 在 Chrome 浏览器中加入了对 H. 264 编解码算法的支持，因此 WebRTC 也支持了 H. 264。

WebRTC 框架由浏览器端(前端)和 Web 服务端(后端)组成，前端用于会话交互、流媒体发送、接收与呈现、语音对讲；后端用于为浏览器端提供 HTML 和 JaveScript 页面，并负责为会话的各方交换通信媒体信息(握手信息)以促成会话各方的点对点直接通信。

2. WebRTC 框架的组网方式

WebRTC 框架具有如下组网方式：

➢ **WebRTC 三角形组网**：会话双方的浏览器互相通信，可以通过一台公共的 Web 服务器进行流媒体握手信息交换，并借此完成 NAT 打洞，其组网结构如图 21 - 12 所示。

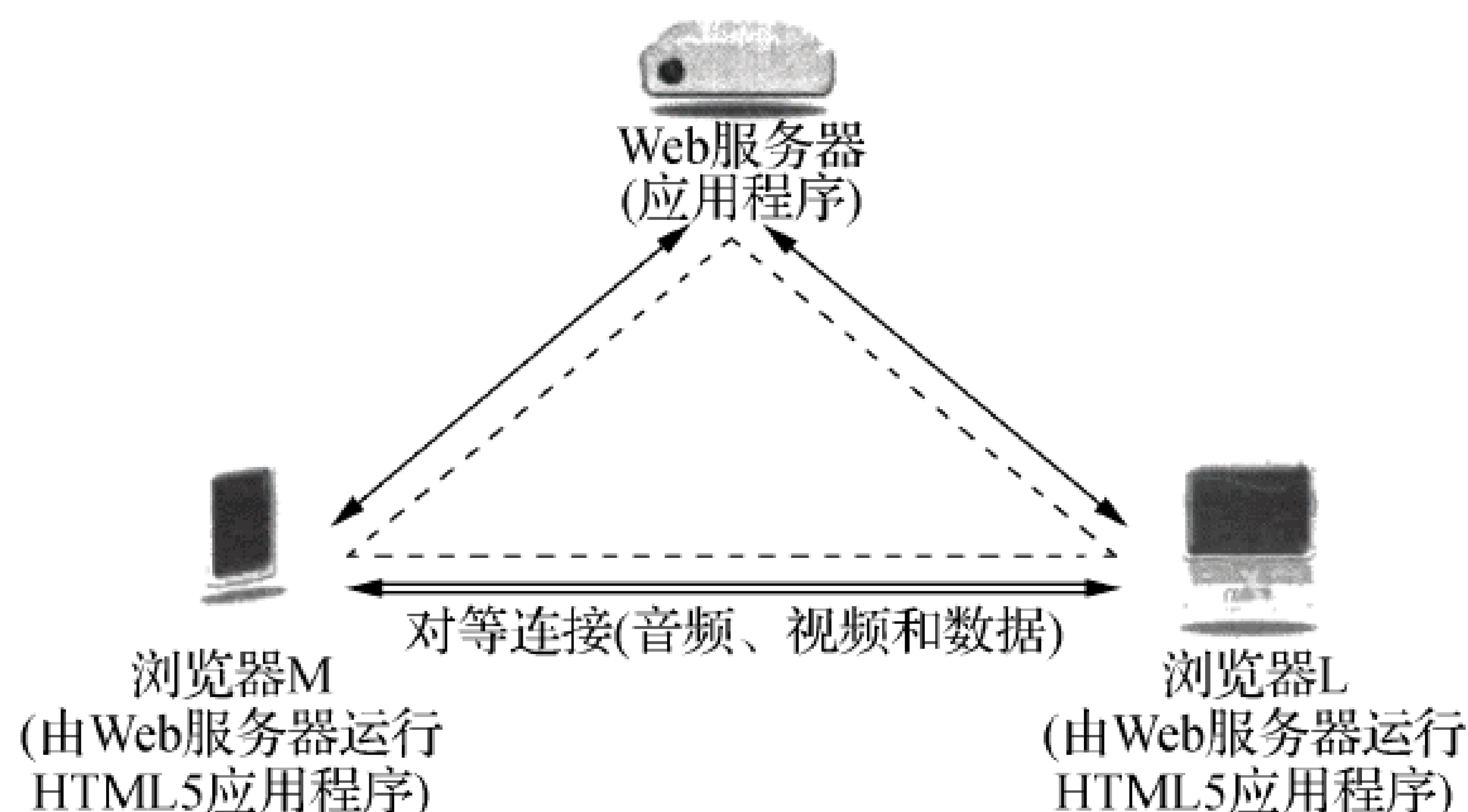


图 21 - 12 WebRTC 三角形组网方式

➢ **WebRTC 梯形组网**：会话双方的浏览器不能仅通过一台 Web 服务器交换信息(例如某一侧的浏览器与另一侧的 Web 服务器之间网络不通)，而需要通过两台 Web 服务器分别作为己方一侧的服务代理来与对方交换媒体握手信息，并基于双方的 Web 服务器提供的打洞信息完成私网穿透，其组网结构如图 21 - 13 所示。

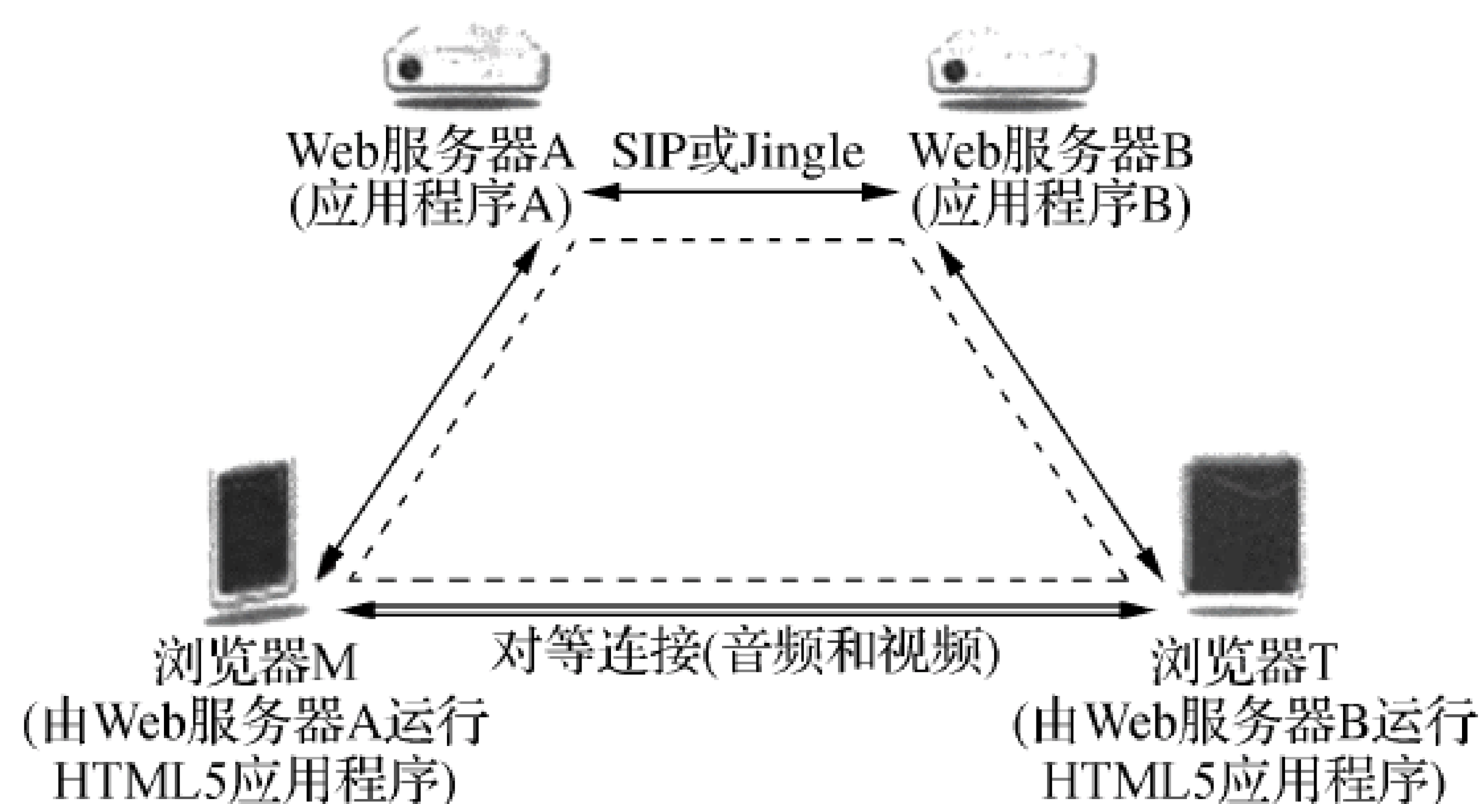


图 21－13　WebRTC 梯形组网方式

➢ **WebRTC 多方会话组网**:会话有时不只有两方,在视频会议场景中经常有多于两方的会话主体,WebRTC 框架也支持多方会话组网方式。与前两种方式类似,该组网方式中也有一个或多个 Web 服务器用于交换握手信息,并且在多方会话主体不能两两之间交互流媒体数据时由媒体服务器负责转发音视频等媒体数据。针对多方会话的场景存在两种组网方案:网格方式和 MCU(多点控制单元)方式,如图 21－14 所示。

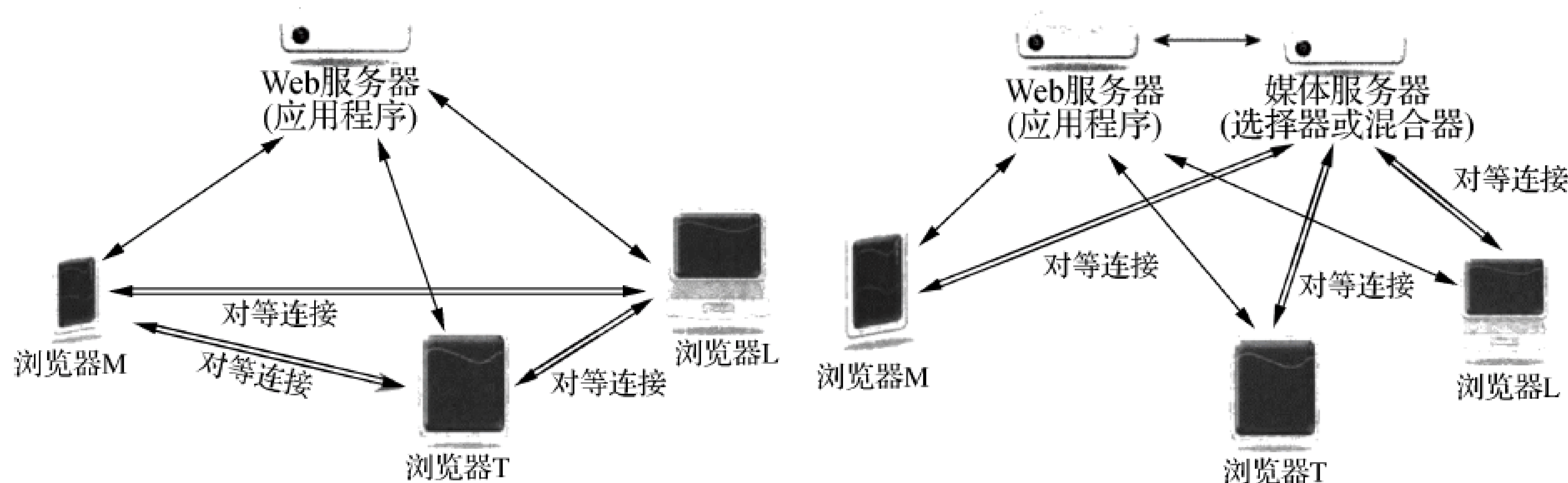


图 21－14　WebRTC 多方会话组网方式

3. WebRTC 框架下的通信过程

使用 WebRTC 框架通信主要包括以下几个步骤:

(1) 通过 MediaStream 方法获取本地媒体。

(2) 通过 RTCPeerConnection 方法在会话的浏览器之间建立对等连接。

(3) 将流媒体和数据通道关联到这条连接上。

(4) 使用 RTCSessionDescription 方法交换会话描述信息。

(5) 会话双方交换媒体数据。

我们以两方会话场景为例,更具体地看一下几个参与方的交互流程:

(1) 会话双方的浏览器分别向 Web 服务器请求会议网页资源(HTML、JavaScript 页面等)。

(2) Web 服务器向双方的浏览器返回操作显示页面和带有 WebRTC JavaScript 代码的页面。

(3) 流媒体会话的发起方(浏览器)将自己的流媒体能力描述以 SDP 的方式发给 Web

服务器,我们称发起方的 SDP 为 offer(请求)。

(4) Web 服务器向流媒体会话的接受方浏览器(WebRTC JavaScript)发送发起方的 SDP 信息。

(5) 流媒体会话的接受方(浏览器)将自己的流媒体能力描述以 SDP 的方式发给 Web 服务器,我们称接受方的 SDP 为 answer(回应)。

(6) Web 服务器向流媒体会话的发起方浏览器(WebRTC JavaScript 页面)发送接受方的 SDP 信息。

(7) 双方的浏览器(WebRTC JavaScript 页面)尝试打洞穿透 NAT 的操作,打洞成功后交换流媒体数据。

(8) 若打洞不成功,则需要借助各自的 Web 服务器或流媒体服务器交换流媒体数据。

图 21-15 和图 21-16 是 WebRTC 三角形组网和梯形组网方式下的会话流程。

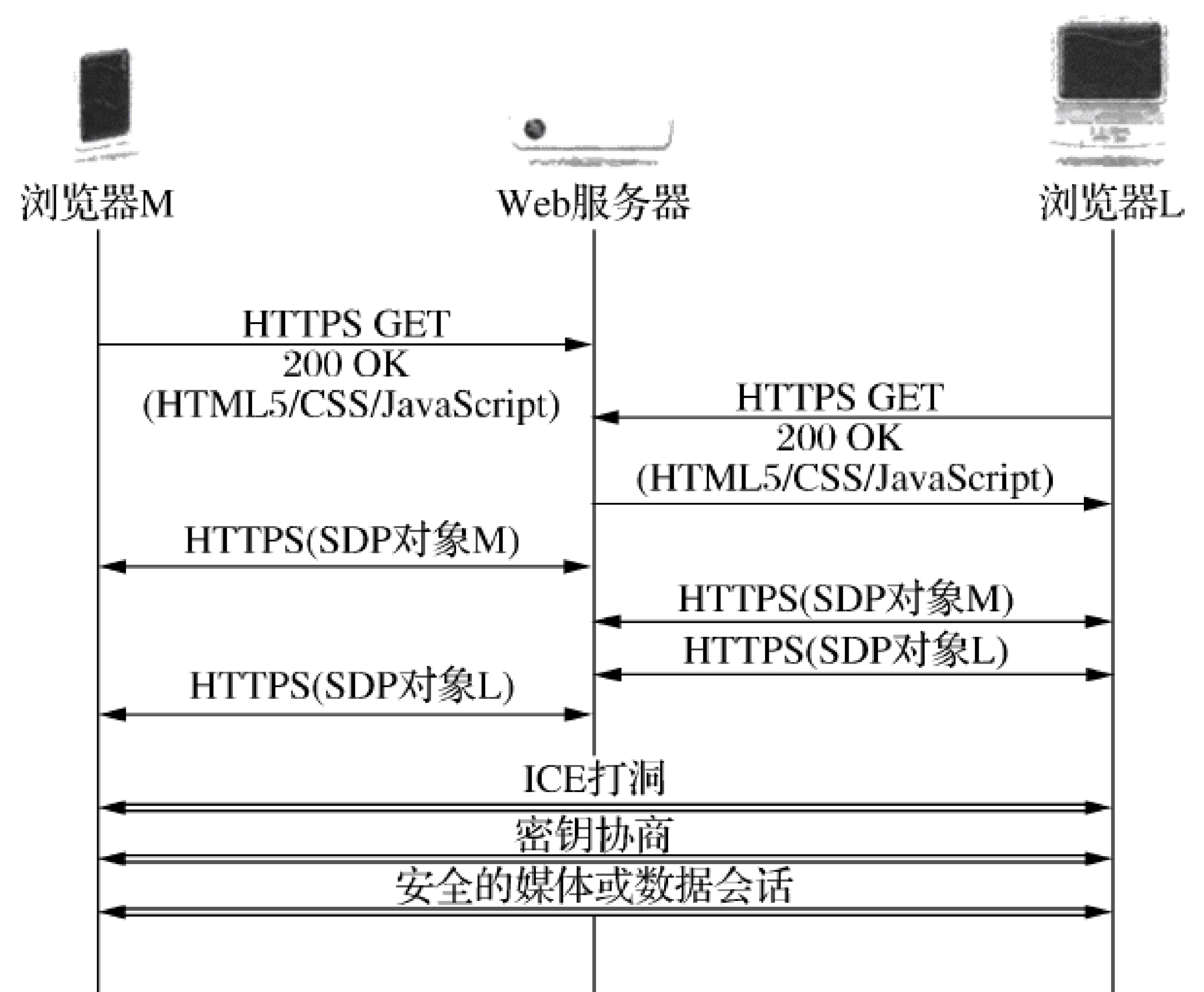


图 21-15 WebRTC 三角形组网的会话流程

视频会议中的媒体包括麦克风音频、摄像头实时视频、共享的视频片段等多种类型。WebRTC 模型采用“轨道”的概念代表一种媒体类型,采用“源”的概念代表媒体流。会话者将“源”分成多个流分发给不同的与会者,每一份流又分为音频和视频两种“轨道”,如图 21-17 和图 21-18 所示。

4. WebRTC 框架的拥塞控制算法

这里还有必要简单提一下 WebRTC 的拥塞控制算法 GCC(Google Congestion Control)。

WebRTC 作为一个端到端的视频通信解决方案,除了要支持尽可能多的编解码算法、简要的 API、强大的可移植性和跨平台特性外,一定要处理好在弱网情况下的发送与接收的问题。特别是应用于视频会议场景下时,弱网是一个正常的存在,因此必须要支持发送码率、发送速率的动态调整,而要实现这些调整,首先得能够判断出当前的网络状态,也就是丢包率和网络延迟。

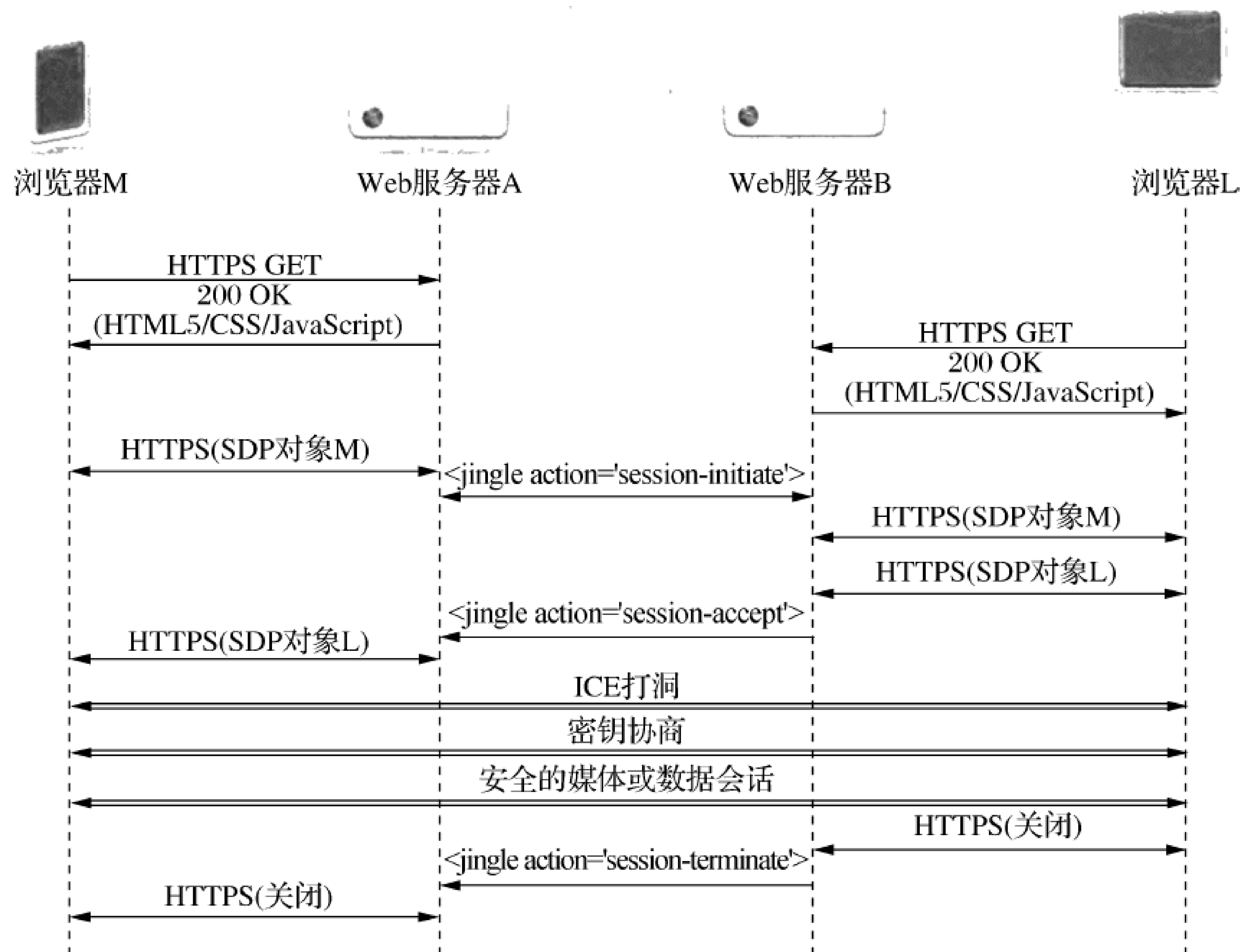


图 21－16　WebRTC 梯形组网的会话流程

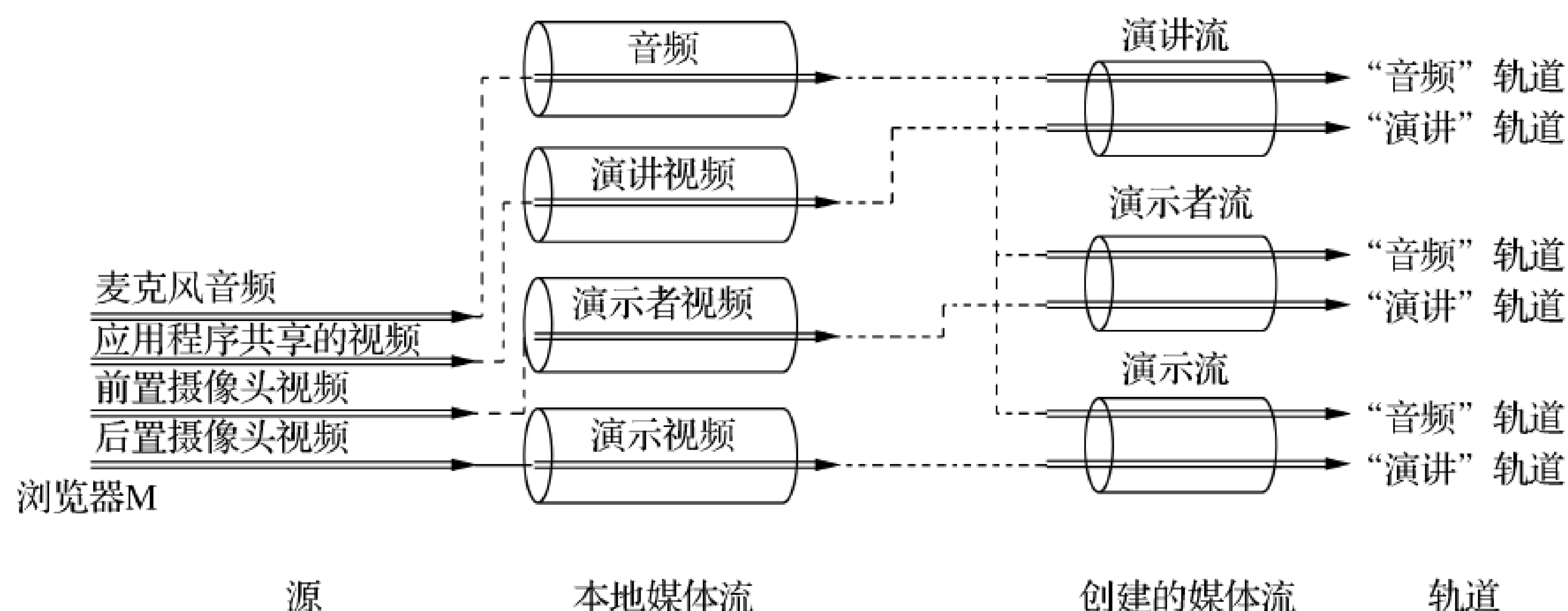


图 21－17　发送者的媒体源与轨道

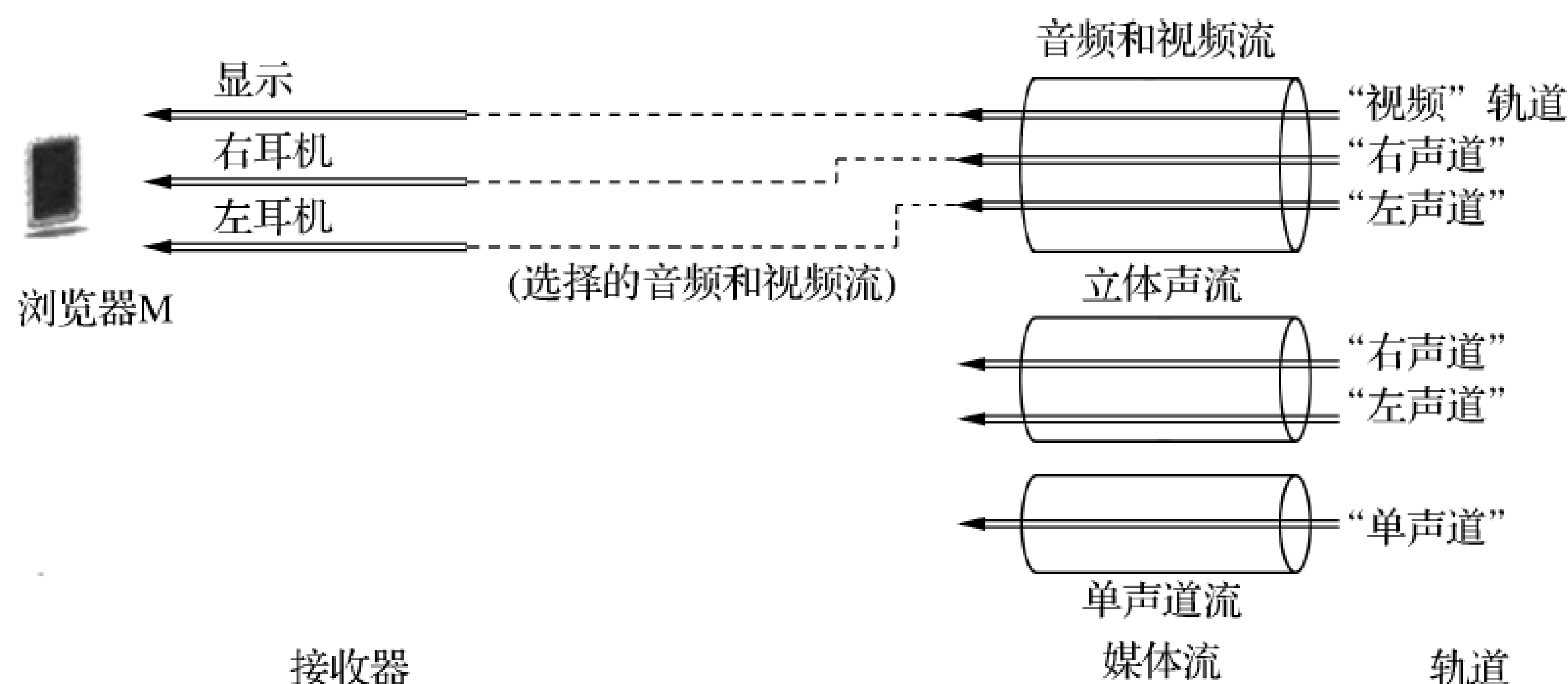


图 21－18　接收者的媒体源与轨道

WebRTC 是通过 RTP/RTCP 协议来反馈丢包率和网络延迟的。为此发送端和接收端各有自己的机制,分别来判断丢包率和网络延迟。

(1) 发送端

发送端基于丢包率来判断网络状态,通常需要接收端采用 RTCP 的 RR(接受者报告)包来进行反馈,在 RR 包的通用 RTCP 头字段后有一个 fraction lost 字段就是专门用于反馈丢包率的。这很简单,也很好理解。

(2) 接收端

接收端是基于延迟梯度来判断网络状态的。那么什么是延迟梯度? 简单地说,梯度延迟本质上是各个数据包接收端收到数据包的时间减去发送端发送数据包的时间后的差值,也就是数据包穿越整条链路过程中耗时的差值。打个比方,如果接收端在接收一个个 RTP 包的时候,每个包的穿越时长都是 10 ms,那么可以认为延迟梯度为 0,从而可以认为这个网络状态比较稳定(当然现实情况中每个包的穿越时长肯定不可能是一样的,一定会有波动,而 WebRTC 通过滤波和组策略平滑掉了这个波动);反之,如果接收到第一个包时发现穿越时长是 10 ms,第二个变成了 15 ms,第三个变成了 20 ms,那么延迟梯度就变成了 5,这说明网络状态是在变差的。当然,梯度值也有可能是负的,这表示网络拥塞情况在逐渐变好,中间传输设备的缓冲区的数据包正在加速消化中。

接收端根据延迟梯度计算出带宽结果后通过 RTCP 的 RR 包的 remb 字段来反馈。同时,由于接收端要计算穿越时长,因此必须要得到该数据包在发送端的发送时间,因此 WebRTC 也扩展了 RTP 头部结构,即增加了一个扩展字段 abs - send - time 来表示 RTP 包的发送时间。

其实,在较新的 WebRTC 版本中,两种拥塞控制策略的计算和研判都挪到了发送端,当然计算的策略和机制都没有什么改变,只是接收端的 RTCP 的 RR 包要包含 RTP 包的到达时间了。

可以看出,WebRTC 的拥塞控制机制的重点还是原先处于接收端的延迟梯度判断。这个判断模块主要由三个组件构成:到达时间滤波器、过载检测器和速率控制器。这三个组件按照上述顺序来处理 RTP 包的到达延时和发送间隔,其主要步骤如下:

- 到达时间滤波器根据各个 RTP 包间的到达延迟和发送间隔计算延迟变化。在这一步骤中还采用了卡尔曼滤波来平滑网络抖动、测量精度和网络噪音等因素造成的误差。
- 过载检测器根据上述延迟变化来判断当前网络状态:overuse(过载)、underuse(使用不足)或 normal(正常)。
- 速率控制器根据上述状态和事先设定的一个阈值来计算带宽估计值。

经过这些步骤,接收端就生成了根据延迟梯度算出的带宽估计值。那么另一侧的发送端也会周期性地根据丢包来算出基于丢包率的带宽估计值,WebRTC 则在这二者之间选取最小值来作为新的发送速率。

21.2.3 Netty 框架

Netty 是由 JBoss 开发的异步 I/O 通信框架,它与 Tomcat 最大的不同之处就是通信协议。Tomcat 是一个基于 HTTP 协议的 Web 容器,但 Netty 却可以通过编程自定义各种协议。因为 Netty 框架中存在协议支持模块,可以通过 Codec 框架编码、解码字节流,完成更高级的业务功能。除此之外,Netty 框架还具有以下特性:

- 并发性能高:采用基于 select 机制的 Reactor 模型。
- 传输速度快:零拷贝,Netty 接收和发送数据采用 Direct Buffers 模式,使用堆外直接内存进行 socket 读写而不需要进行字节缓冲区的二次拷贝。
- 安全加密:支持 SSL/TLS/StartTLS。
- 提供对多种编解码框架的集成:包括 Protobuf、Jbossmarshalling、Java 序列化、压缩编解码、XML 解码、字符串编解码等编解码框架可以被用户直接使用。
- 可以对网络事件进行拦截和过滤。
- 是当下非常流行的 Java 通信框架:封装简单,使用方便,社区活跃。

我们先来看一下 Netty 的功能框架,如图 21-19 所示。

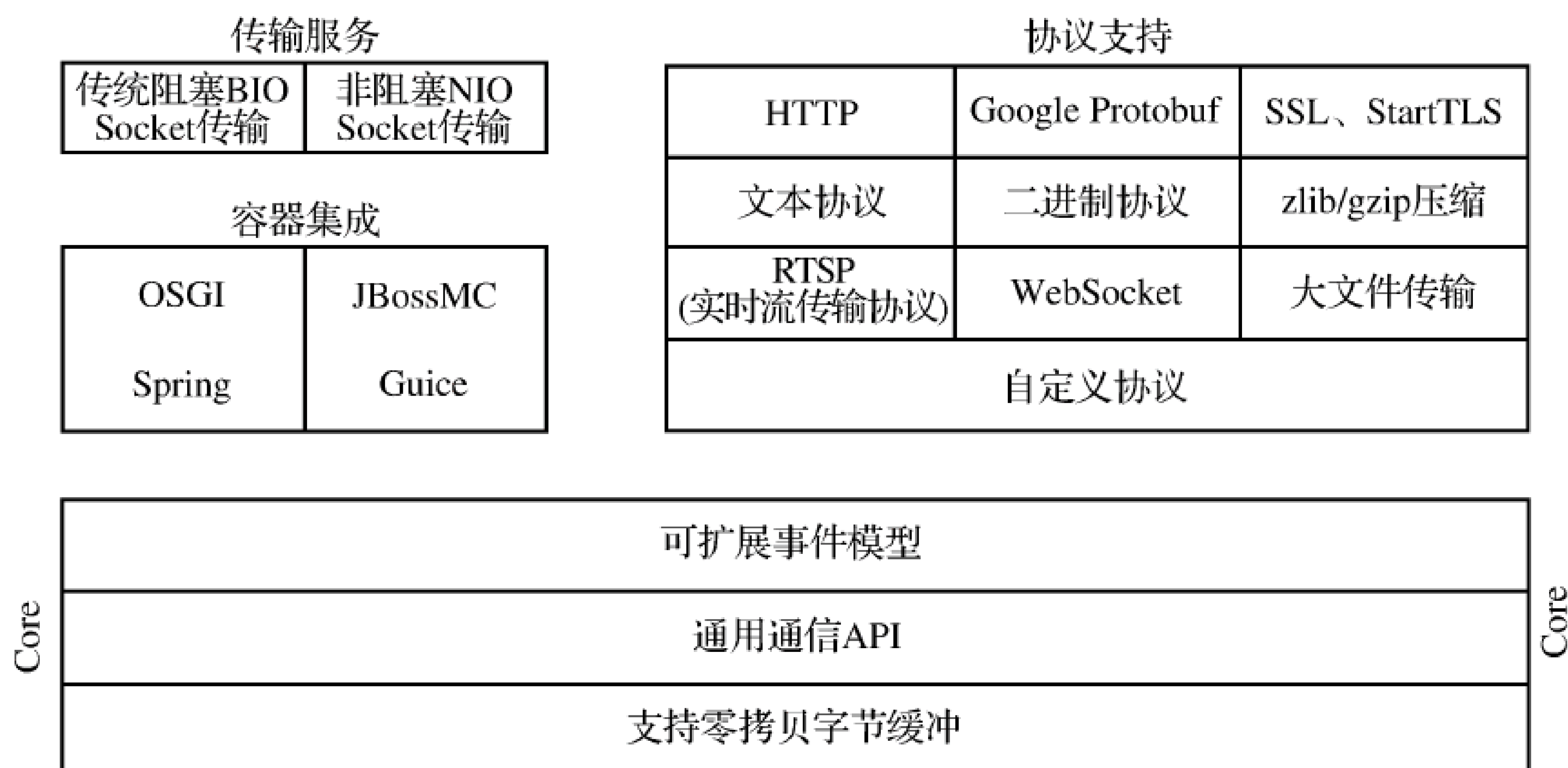


图 21-19 Netty 框架功能框架视图

Netty 改进了 Reactor 模型,将 Reactor 分成 mainReactor 和 subReactor 两部分。以 TCP 服务端为例,mainReactor 负责 TCP 服务端的 socket 监听与接受,并将接受的 socket 分派给 subReactor;subReactor 负责事件多路分离,并处理所有的读写事务和业务逻辑功能,如图 21-20 所示。

在 Netty 中,mainReactor 就是 Boss 类,而 NioWorker 类则充当 subReactor,且 NioWorker 类的个数与当前系统中 CPU 的核数相当。以 TCP 服务端为例,Boss 对象专门负责监听 socket 并接受 TCP 客户端发过来的连接请求,但不处理读写事务;同时,若干个 NioWorker 对象则专门读写数据,处理业务逻辑,这种模型具有很高的并发效率。

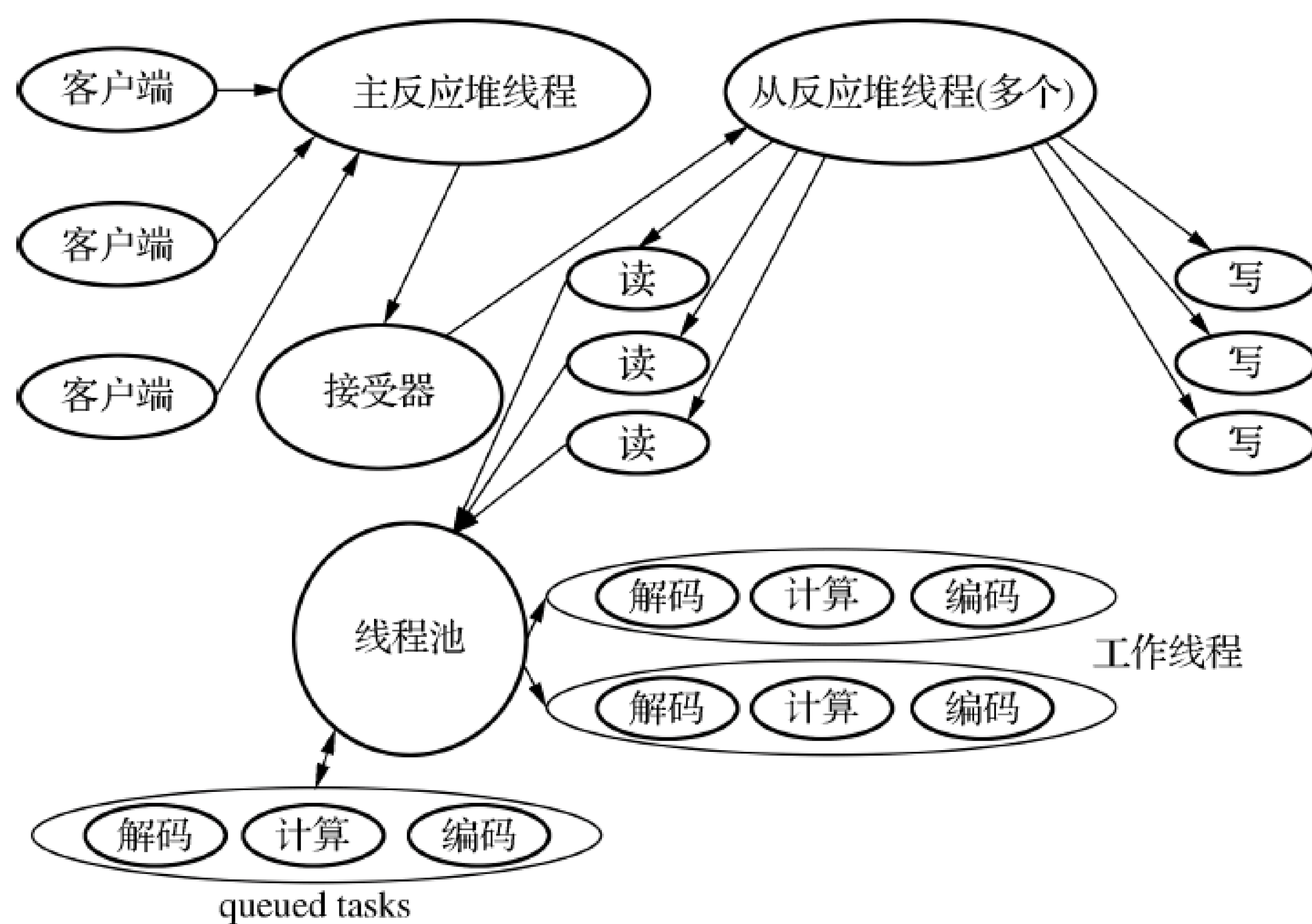


图 21－20　Netty 框架通信模型视图

在 Netty 中，Channel 表示一个连接；ChannelHandler 用于数据收发处理业务；ChannelHandlerContext用于在 Channel 之间传输业务数据；ChannelPipeline 则保存各处理过程的 ChannelHandler 和 ChannelHandlerContext。

面向 Channel 的数据读写是 Netty 的一大特点。ChannelPipeline 中各组件之间的协作关系如图 21－21 所示。

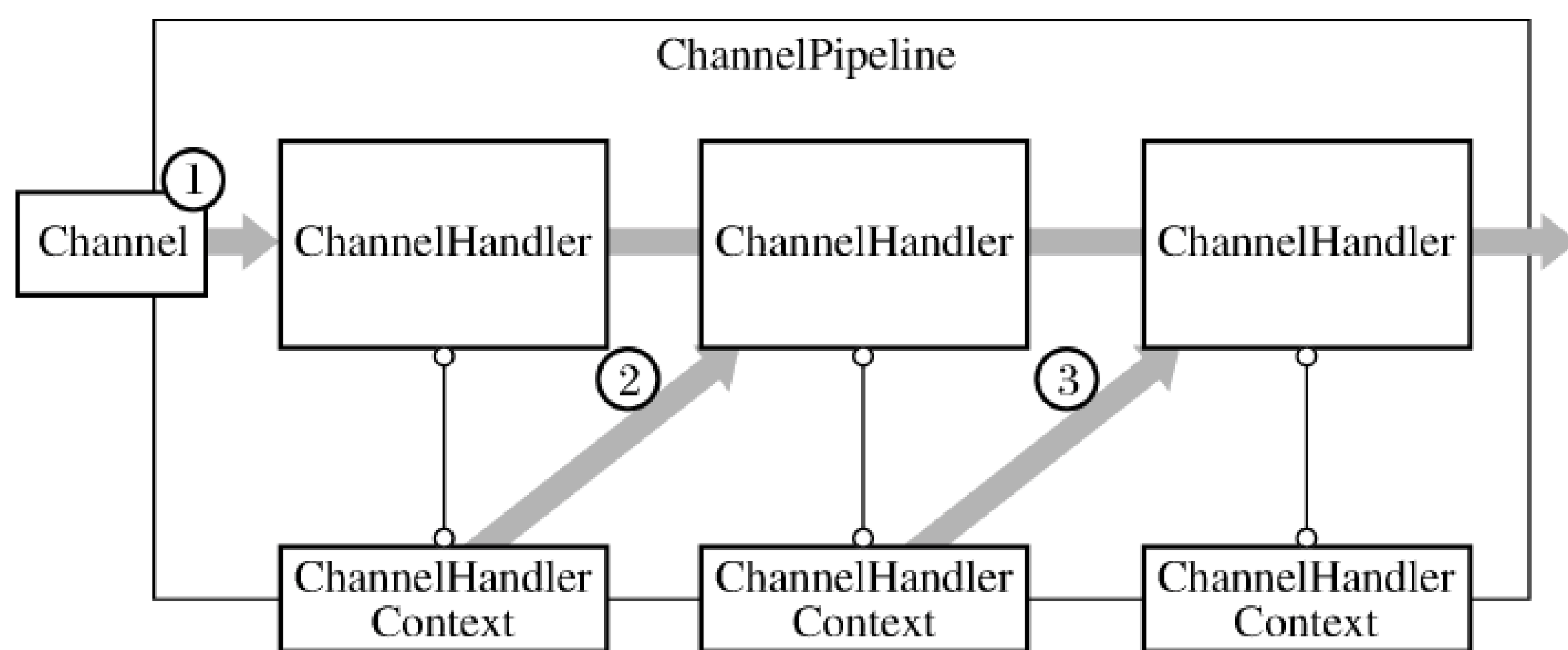


图 21－21　ChannelPipeline 中各组件的协作关系

不同协议、不同阻塞类型的连接都有不同类型的 Channel 与之对应，这些 Channel 涵盖了基于 UDP/TCP 的网络 I/O 以及文件 I/O 操作，包括：

- **NioSocketChannel**：异步的 TCP 客户端连接；
- **NioServerSocketChannel**：异步的服务器端 TCP socket 连接；
- **NioDatagramChannel**：异步的 UDP 连接；
- **NioSctpChannel**：异步的 SCTP 客户端连接；
- **NioSctpServerChannel**：异步的 SCTP 服务器端连接。

Netty 是由事件驱动的，通过 ChannelPipeline 来控制事件流，各个 ChannelHandler 被注册

到 ChannelPipeline 上处理数据读写事务。事件流分为两种,上行事件(upstream)处理外发,下行事件(downstream)处理接收,因此注册到 ChannelPipeline 中的处理器 ChannelHandler 可以是 ChannelUpstreamHandler 或 ChannelDownstreamHandler。它们可以终止流程,也可以将事件(ChannelEvent)传递下去,传递流程可以通过 ChannelHandlerContext 的 sendUpstream 或 sendDownstream 方法实现。

21.2.4 Service Mesh

1. Service Mesh 简介

Service Mesh(服务网格)最早是由 Buoyant 公司提出的。这是一个轻量级高性能网络代理框架,服务于云原生的应用,特别是微服务场景下的通信管理。因此 Service Mesh 是介于三层和四层协议之间的网络代理的抽象框架,是云服务中的基础设施层。

与前述各通信框架不同,Service Mesh 是瞄准微服务集群场景下的分布式通信框架,而非像 ACE 等框架一样针对的是单体进程的通信服务。

在微服务场景下,单个应用程序可能包含几十甚至上百个单体服务实例,每个服务实例都处于不断变化的状态(例如处于 Kubernetes 调度环境中时),对于网络流量的引流、熔断、加密、负载均衡、网络监管等基础设施和运维功能如果都由业务模块自己实现,则会造成庞大的开发量和较高的开发门槛。Service Mesh 就是在这种诉求之下产生的,其目的是将通信服务从上层业务和底层系统中剥离出来形成一个单独的框架,以方便对网络和流量进行监管和控制,同时解放上层业务网络管理的束缚,使业务模块的开发门槛降低,工作量变小。

Service Mesh 的典型组网方式如图 21 - 22 所示。

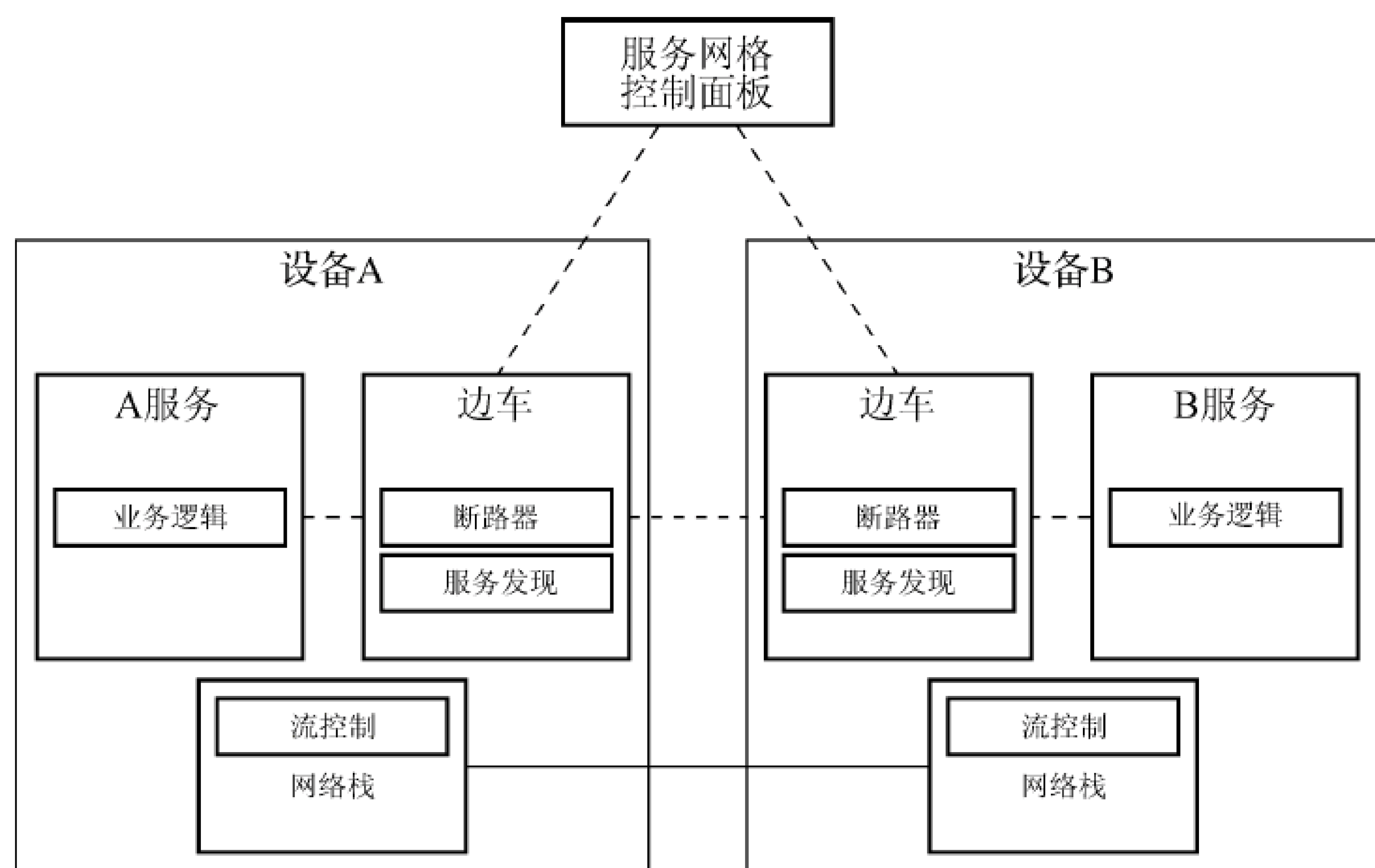


图 21 - 22 Service Mesh 的典型组网方式

Service Mesh 一般分为数据平面和控制平面两部分,这与 SDN(软件定义网络)的核心思想不谋而合。

数据平面负责具体通信业务,包括数据传输、数据包转发、路由、服务发现等,一般采用

通信代理的方式与业务模块交互,我们将这个代理称为 Sidecar(边车,一个比较莫名其妙的名字)。Sidecar 负责具体的通信实现(TCP、RPC、HTTP 等)。例如在 Kubernetes 中使用 Istio(Service Mesh 的一个落地项目)时,应用进程所在的容器通过 Pod 中共享的网络命名空间内的 Loopback 接口与 Sidecar 通信,这对于其他 Pod 和节点代理是不可见的。Linkerd 和 Envoy 等开源项目是 Service Mesh 数据平面的具体落地实现,本节中我们会以 Envoy 为例讲述数据平面。

控制平面负责策略下发、熔断管理、流量加密,并承担数据平面的转发业务的管理工作。这个"管理"的范围包括下发转发策略、流量策略等,但不包括直接转发数据包,同时要承担与外部系统的接口和网络行为可视化等工作,这部分是 Service Mesh 的核心。

与 SDN 类比,Service Mesh 的控制平面相当于 SDN 控制器,但 Service Mesh 的数据平面却不会将不能处理的数据包汇报上交给控制平面,这一点与 SDN 的转发平面是不一样的。

2. Istio

Istio 是 Google、IBM 和 Lyft 联合开发的开源项目,是 Service Mesh 框架的具体落地产品。Istio 于 2017 年 5 月发布了第一个 Release 版本,其官方的定义是:一个连接、管理和保护微服务的开放平台,是一种将网络层委托给 Istio 的服务网格。

Istio 的设计初衷是为了将网络管理与业务模块分离开来以使单个微服务的复杂度降到最低,而 Istio 超越 Spring Cloud 和 Dubbo 等框架的地方就在于其提供了远超过后者的功能集合,并且不需要应用程序做多大改动,开发的门槛也比较低,比较全面地解决了微服务治理中的诸多问题。

Istio 遵从 Service Mesh 的基本分层思想,亦分为数据平面和控制平面两部分。数据平面由网络代理 Envoy 组成,部署形态为 Sidecar,Sidecar 用于调节和控制微服务之间的所有网络通信。控制平面由 Pilot、Mixer 和 Istio-Auth 构成,负责管理流量的路由以及运行时的执行策略。Istio 的整体架构如图 21 - 23 所示。

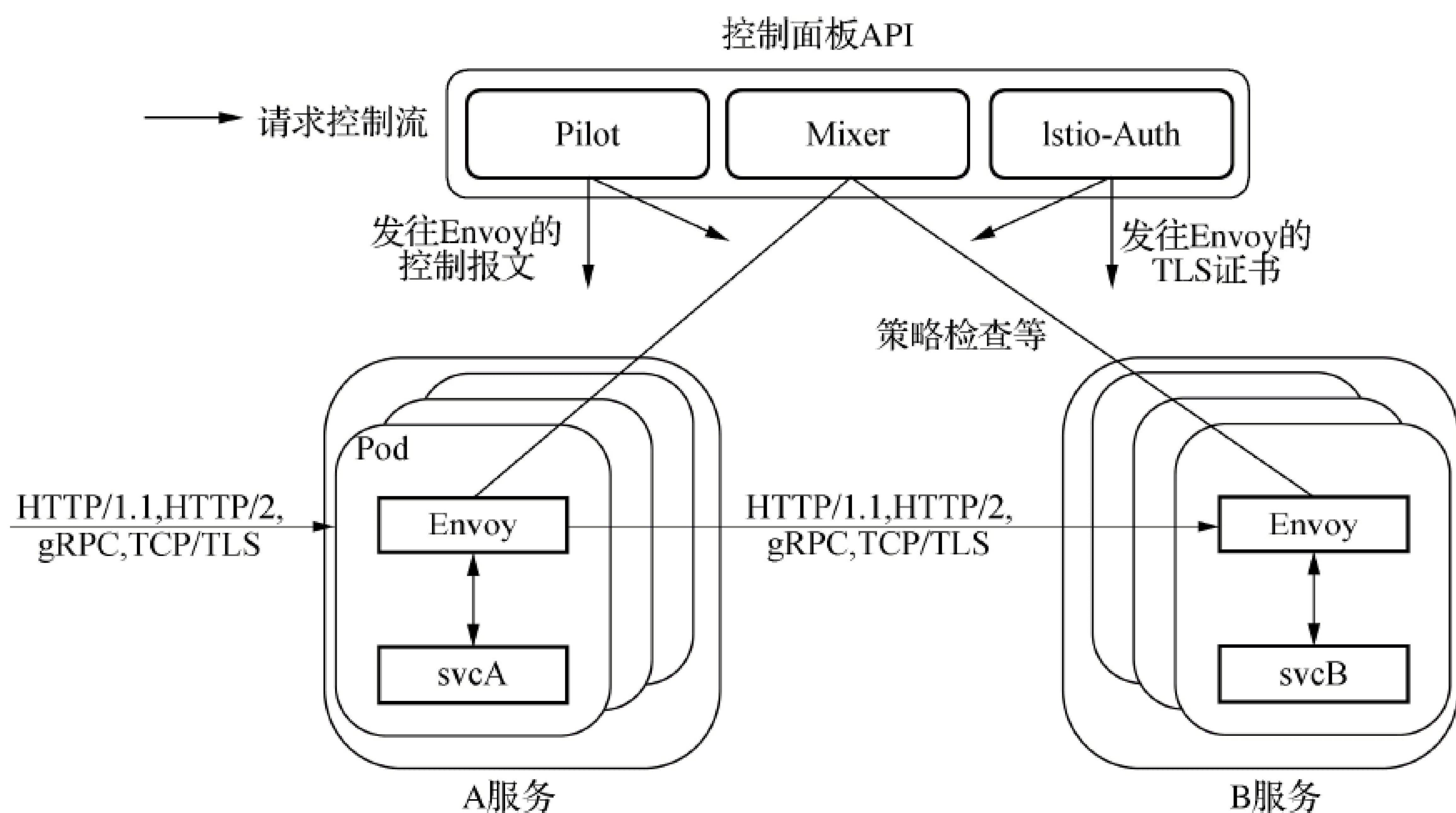


图 21 - 23　Istio 架构视图

3. Pilot

Pilot 是 Istio 控制平面中负责流量管理的组件。至于其中文名称，总不能翻译成“飞行员”，但翻译为“导航器”还是比较贴切的，因为流量管理的具体内容就是路由请求、服务发现、负载均衡、故障处理、故障注入、规则配置等这样一些与流量导引有关的工作。

Pilot 是 Envoy 的管理者，它定义了一个抽象模型（Abstract Model），以便从特定的平台细节中解耦出来，提供跨平台能力（如 Kubernetes 或 Mesos 等）。Pilot 的框架结构如图 21－24 所示。

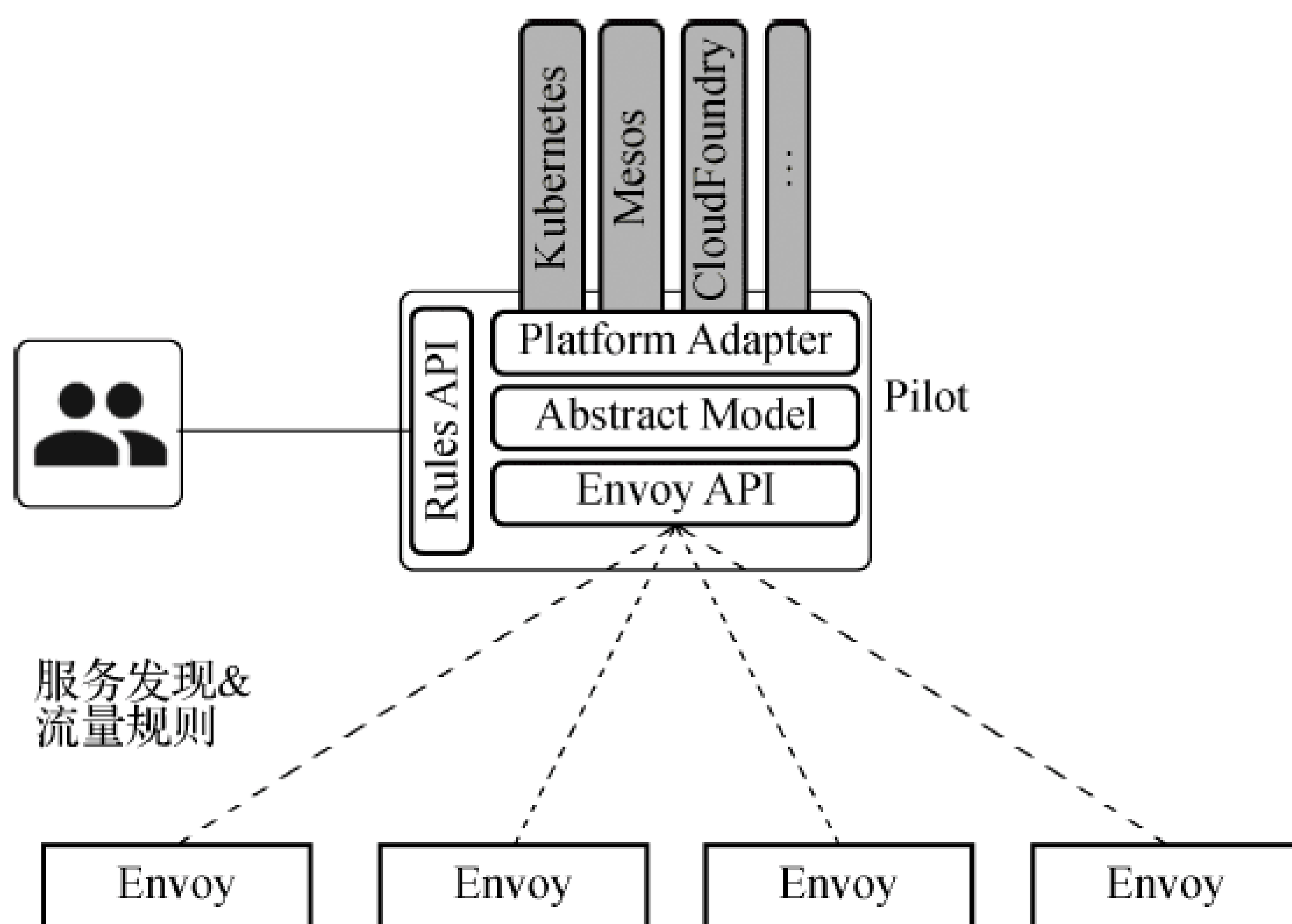


图 21－24　Pilot 框架结构（图片来自 CSDN）

Pilot 框架包含以下组件：

- **Envoy API 组件**：作为与 Envoy 交互的第一层逻辑实体，负责与 Envoy 之间所有的通信，包括向 Envoy 下发服务发现信息、流量控制规则等。
- **Platform Adapter 组件**：用于对接不同的外部平台（如 Kubernetes 或 Mesos 等），它是跨平台抽象模型的具体实现。
- **Rules API 组件**：向外部平台提供北向接口以供第三方软件管理 Pilot。
- **Abstract Model 组件**：是对服务网格中“服务”的规范表示，用于定义在 Istio 中什么是服务，这个规范独立于底层平台。

4. Mixer

Mixer 一般被翻译为“混合器”，是在服务网格中执行访问控制和使用策略的组件，也是被 Envoy 重度依赖的组件。Mixer 具备以下功能：

- 收集 Envoy 代理和其他服务报送的遥测数据，例如服务日志和实时监控信息等。
- 检查前提条件，对上层应用软件传过来的请求进行验证，例如权限认证、黑白名单验证和 ACL 检查等。
- 管理配额，可以在多个维度上分配和释放资源，提供限速等配额类控制服务。

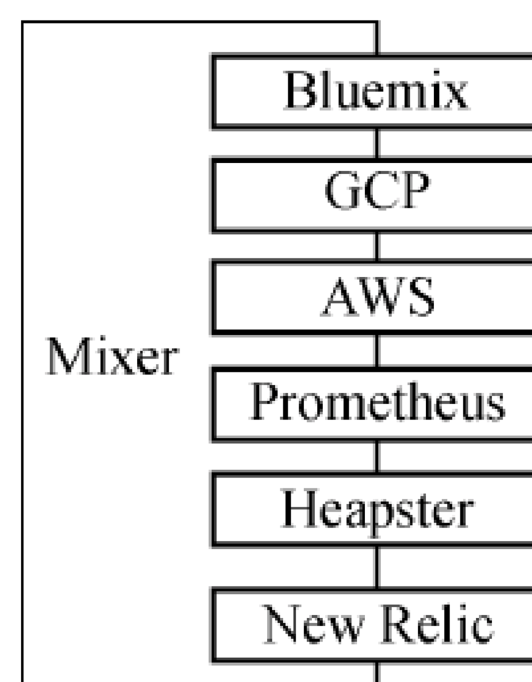


图 21－25　Mixer 的适配器设计思想

Mixer 采用适配器插件的方式提供 Istio 与不同后端服务之间的对接,这些后端服务包括计费、日志管理、运维、状态监控等业务,这种设计思想与各后端的服务对接,使得另一侧的接口保持不变,做到了后端无关化(如图 21 - 25 所示)。

5. Istio-Auth

Istio-Auth 组件用于提供服务到服务之间的用户认证服务,也可用于加密服务网格中的流量,例如数据平面 Envoy 之间的通信就是采用了 TLS 方式进行加密的。Istio-Auth 的架构如图 21 - 26 所示。

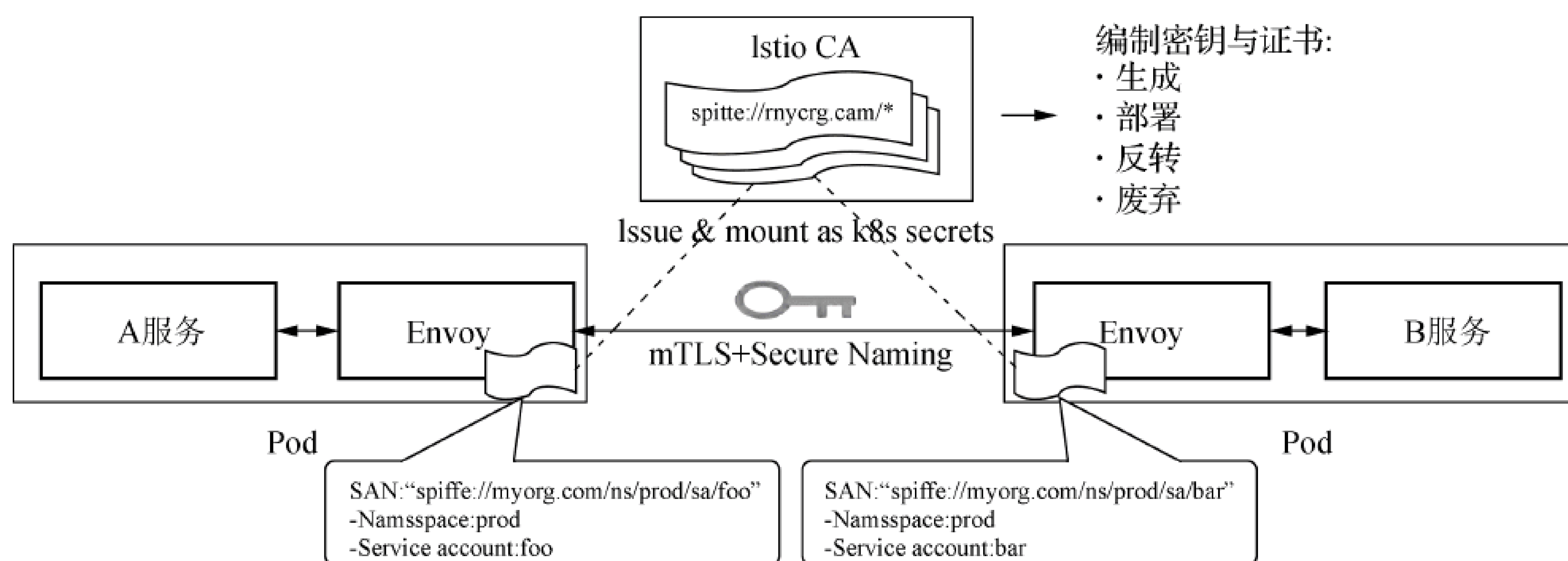


图 21 - 26　Istio-Auth 架构视图(图片来自 CSDN)

6. Envoy

Istio 作为 Service Mesh 的具体落地项目,使用的数据平面是 Envoy 的扩展版本。

Envoy 是由 C++ 语言开发的高性能网络通信代理框架,为各个服务提供了基础网络通信能力,包括 HTTP1.1、HTTP2.0、gRPC(由 Google 开发的一款语言中立、平台中立、开源的远程过程调用系统)和 TCP 等。除了支持上述四层或五层通信协议,Envoy 也可以调节服务网格中所有服务的出入站流量。除此之外,Envoy 的能力还包括:

- 服务发现:可以接收 Istio 中 Pilot 组件的服务发现信息。
- 断路器:这是一种服务自我保护的机制,当某个服务节点因故障宕机时,Envoy 可以检测到该节点出入流量的异常,若判定为故障,后续的流量不再调度到该故障节点。
- 负载均衡。
- 健康检查。
- 故障注入:故意引入故障,以扩大测试范围来探测故障边界,提升系统的健壮性。
- 执行路由规则:即接受 Istio 中 Pilot 组件的路由和目的地策略。
- 加密和认证:保证安全传输。
- 为 Istio 中的 Mixer 组件提供数据。

除了 Envoy 外,还存在其他数据平面的落地项目,如 Linkerd 组件。这是一款基于 Scala 语言开发的通信组件,可以与 Istio 的控制平面对接以替代 Envoy。

7. Istio 的应用

Service Mesh 最常用的场景是云化的容器环境，特别是基于 Kubernetes + Docker 的容器化环境监控网络服务状态，并基于此为上层业务服务。

其实，Kubernetes 已经提供了诸如 Service 资源对象管理、负载均衡这样的基础功能，但为何还要以 Service Mesh 作为云应用程序的基础功能设施组件呢？

一般来说，在大部分应用软件中，上述基础功能逻辑直接构建在程序本身的各层软件栈中了，例如超时重连机制、服务监控机制、服务发现机制等。但是随着应用程序架构越来越微服务化，将底层的通信相关的业务逻辑从应用程序中剥离出来就变得越来越必要了。也就是说，应用程序只需要关注自己的业务实现，而不应该涉及通信服务，不应该关注网络协议栈，也不应该负责自己的负载均衡和熔断机制。基于这种考虑，将这些共性的、底层的业务逻辑剥离和抽象出来作为公共服务组件就成了必然。特别是在云化环境中，在容器中部署微服务组件成了大趋势，轻量级的微服务业务模块非常需要这些底层公共组件的支撑。这些底层公共组件也被称为“服务治理”组件。

Kubernetes 的 Pod 里可以包含多个容器，同一个 Pod 的多个容器之间共享卷、网络和 IPC（进程间通信机制）。Kubernetes 的 Pod 机制给我们提供了一种能力，就是将一个本来要捆绑在一起的服务拆成多个，可以分为主容器和副容器（Sidecar），这是一种更细粒度的服务拆分能力。

在 Istio 与 Kubernetes 结合使用的场景下（如图 21－27 所示），Istio 在业务 Pod 里部署了一个 Sidecar（Envoy 或 Linkerd），这是一个代理服务器，前文提到的网络层功能基本依靠它来实现，Envoy/Linkerd 和上层的控制层组件（Mixer、Pilot、Istio-Auth）交互，实现动态配置、策略执行、安全证书获取等功能，同时对用户业务透明。

21.2.5 其他通信框架

除了 ACE 和 WebRTC 框架，还有许多其他框架，但由于这些框架的兼容性、普适性、跨平台性、通信效率等均比不上 ACE 或 Netty，因此在这里仅仅做一些简单介绍。

21.2.5.1 Boost. Asio 框架

Boost. Asio（简称 Boost）是个由 C++ 语言编写的轻量级跨平台开源通信框架，依赖于 Boost 库。与 ACE 类似，Boost 框架在 Windows 下基于 IOCP 模型、在 Linux 下基于 epoll 模型实现了异步事件机制。在使用时只需要 include 头文件而不需要导入其他第三方动态库。

Boost 框架自底向上依次是操作系统适配层（由 Boost. System 库提供操作系统支持）、模板类、模板类的参数化和唯一的服务框架 io_service（用于与操作系统打交道，等待所有异步操作的结束）。但 Boost 框架比 ACE 框架的门槛要低得多，且只涉及 socket 通信、COM 串口通信、文件 I/O 和一些简单的线程操作。其中 Boost 框架的事件分派是基于函数对象的 Hanlder 事件分派机制，且任何函数都可能成为 Hanlder。

Boost 框架可以同时支持数千路 TCP 并发连接，其性能接近 ACE 框架。

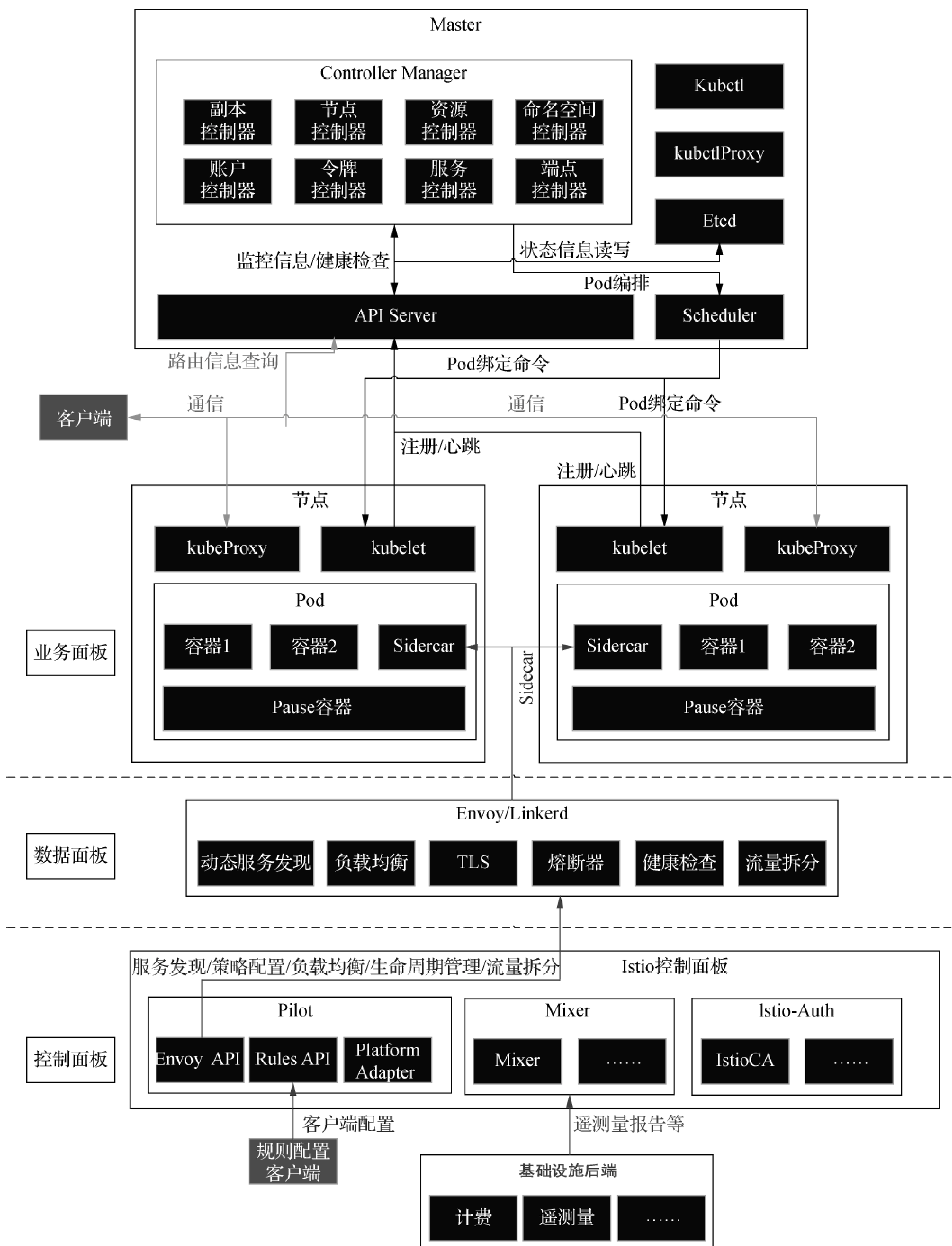


图 21－27　Istio 与 Kubernetes 框架结合使用的场景

21.2.5.2 Libevent

Libevent 是由 C 语言编写的基于事件驱动机制的轻量级通信库,适用于 Windows 和 Linux 等多种平台,其多路分离机制可以基于 IOCP、epoll、select 等多种模型。Libevent 库本质上是一个反应堆框架,向反应堆注册接口和事件,当事件发生时,注册的接口被回调。这些被激活的事件存放在优先级队列中,每个事件的默认优先级都是相同的,但可以调整以提高事件优先级使其被优先处理。

我们借用 ACE 框架中的反应堆来类比 Libevent:

- Libevent 的事件多路分离机制仍然是 select、epoll 等,这与 ACE 框架是一致的。
- 在 Libevent 中仍然要将事件和接口注册到反应堆上,这一点与 ACE 框架也是一致的。
- Libevent 的事件处理器是 Event 结构体,对应了 ACE 框架中的 ACE_Event_Handler 类。

Libevent 支持的事件及属性包括:

- **EV_TIMEOUT**:表示超时事件;
- **EV_READ**:表示只要网络缓冲区中还有数据未被读出,回调函数就会被触发;
- **EV_WRITE**:表示只要传递给网络缓冲区的数据被上层应用方法写完,回调函数就会被触发;
- **EV_SIGNAL**:表示 POSIX 信号量;
- **EV_PERSIST**:若指定该属性,回调函数被触发后事件不会被删除,反之则会被删除;
- **EV_ET**:表示边缘触发方式。

Libevent 框架的运行流程如下所示:

(1) 调用 **event_init** 方法创建 event_base 对象,一个 event_base 对象对应一个 reactor 实例。

(2) 创建具体的事件处理器并设置它们关联的反应堆。

- **evsignal_new** 和 **evtimer_new** 分别用于创建信号事件处理器和定时事件处理器,它们的统一入口是 event_new 方法;
- **event_new** 方法成功时返回一个 Event 类型的对象,也就是 Libevent 的事件处理器。

(3) 调用 **event_add** 方法将上述返回的事件处理器添加到注册事件队列中,并将该事件处理器对应的事件添加到事件多路分离器中。

(4) 调用 **event_base_dispatch** 方法来执行事件循环。

(5) 事件循环结束后,使用 * **_free** 系列方法释放系统资源。

Libevent 的运行流程如图 21-28 所示。

21.2.5.3 Libev 框架

Libev 对 Libevent 做了优化修改,设计更简练,性能也有所提升,在 Linux 下可支持 epoll 和 KeQueue 机制,但对于 Windows 系统的支持不好,不支持 IOCP 模型。与 Libevent 一样,Libev 也可以设置事件处理优先级。

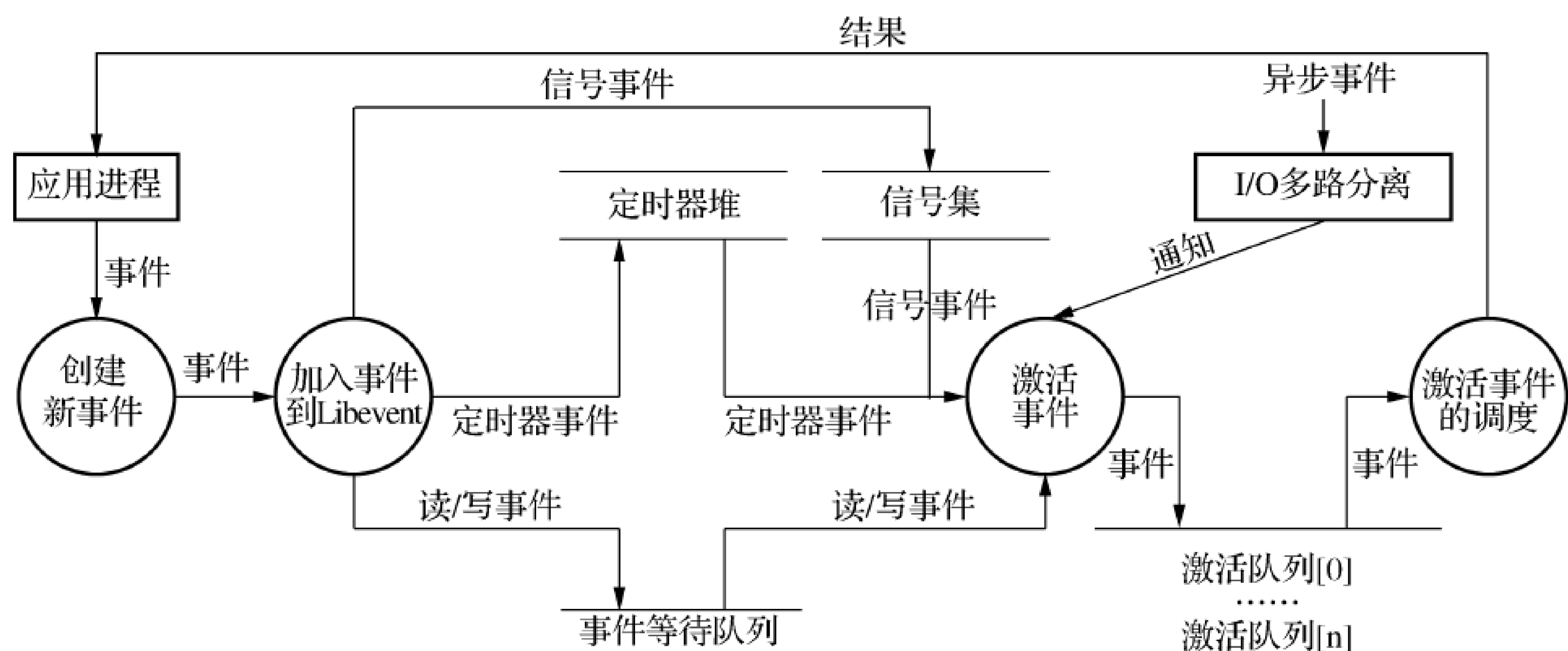


图 21－28　Libevent 运行流程

21.2.5.4　Libuv 框架

Libuv 是 node.js(基于 Google Chrome 的 JavaScript 引擎的 Web 应用程序框架)采用 C 语言封装的轻量级通信库,node.js 依赖的通信库就是 Libuv。Libuv 在 Windows 下使用 IOCP 模型,而在 Linux 下集成了 Libev。Libuv 的特点是使用了大量的回调机制实现异步 I/O 的完成通知。

21.2.5.5　RPC 框架

RPC(Remote Procedure Call,远程过程调用)是一种跨主机、跨进程的通信机制,其特点是屏蔽了通信协议的细节,而将通信请求封装成接口以便于分布式程序的调用。因此,我们称之为“远程过程调用”。

前面的章节中我们介绍过本地过程调用(LPC)。RPC 与 LPC 相对应,只是 RPC 的应用场景更广泛,LPC 的大多数功能都可以由 RPC 代劳。不过两者的实现机制确实大相径庭,LPC 采用共享内存区和端口对象来实现,RPC 则是彻头彻尾地基于 TCP/IP 通信。因此,RPC 分为客户端和服务端,我们定义发起请求的一方为客户端,响应请求的一方为服务端,二者之间采用同步方式通信。在微服务化日益广泛的分布式架构场景中,RPC 有着广泛的应用。

RPC 框架由以下几部分组成:

- **RpcServer**:负责导出(export)远程接口。只有服务端导出了接口,客户端才知道怎样调用。
- **RpcClient**:负责导入(import)远程接口的代理实现,使客户端像调用本地方法一样去调用远程接口方法。
- **RpcProxy**:远程接口的代理实现,RpcClient 与其进行交互。
- **RpcInvoker**:表示一种机制,即客户端负责编码调用信息和发送调用请求到服务端,并等待调用结果返回(当然 RPC 机制也可以改进以做到异步调用,但目前仍以同步调用为主);服务端负责调用服务接口的具体实现并返回调用结果。

- **RpcProtocol**：负责协议编解码。
- **RpcConnector**：负责发起客户端到服务端的连接通道，并发送数据到服务端。报文的传输方式一般采用基于 TCP 的 HTTP 协议。
- **RpcAcceptor**：负责接收客户端请求并返回请求结果。RpcConnector 与 RpcAcceptor 之间通过心跳保活机制以维持连接。
- **RpcProcessor**：负责在服务端控制调用过程，包括管理调用线程池、超时时间等。
- **RpcChannel**：数据传输通道。

RPC 调用流程如图 21－29 所示。

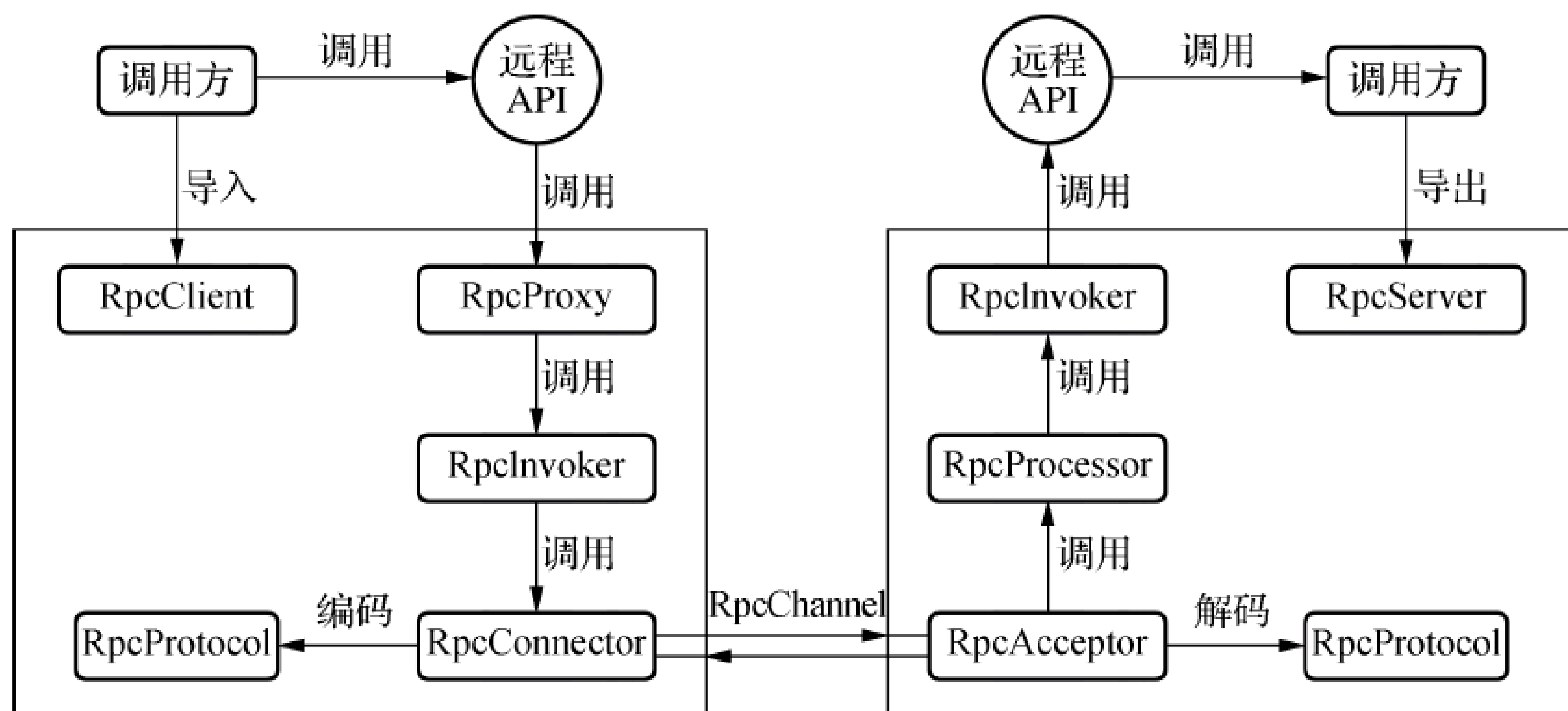


图 21－29　RPC 调用流程

综上所述，RPC 适合同步调用的场景，或者是希望以 API 调用这种开发门槛较低的方式交互的场景。我们比较熟悉的 WebService 调用、Thrift 框架等都是基于 RPC 机制的。但由于是同步调用，所以服务端的处理能力也会直接影响客户端的响应效率。

RMI（Remote Method Invocation，远程方法调用）的本质是 RPC 在 Java 语言上的一种实现：在一个 Java 虚拟机上的对象调用另一个 Java 虚拟机上对象的方法（两个虚拟机可以是远程的或本地的）。因为 RPC 机制是与语言和操作系统无关的，而 RMI 只能用于 Java 语言，其核心是远程 Java 对象，因此 RMI 的使用场景比较特定，不支持非 Java 语言开发的应用进程，也不能与非 Java 语言开发的应用进程进行通信。

RMI 使用 JRMP（Java Remote Method Protocol，Java 远程方法协议）进行通信，通过在客户端的 Stub 对象作为远程接口进行远程方法的调用，这也是要求通信双方必须采用 Java 语言开发的原因。RMI 的调用结果统一由 XDR（External Data Representation，外部数据表示）语言来表示，这种语言抽象了字节序类型和数据类型之间的差异。

客户端只与代表远程主机中 Java 对象的 Stub 对象进行通信，但客户端不知道对端服务端的存在，客户端只是调用本地 Stub 对象中的方法，如图 21－30 所示。Stub 对象是一个本地存根对象，它实现了远程对象公布的接口，也就是说 Stub 对象中的方法和远程 Java 对象暴露的方法的签名是相同的。客户端认为它调用了远程对象的方法，实际上调用的是 Stub 对象中的方法。可以这么说：Stub 对象是远程对象在本地的一个代理，当客户端调用 Stub

对象中的方法时，Stub 对象会将调用通过网络传递给远程对象。



图 21－30　RMI 通信模型

21.2.5.6　HP-Socket 框架

HP-Socket 是一套高性能四层/五层协议通信框架，支持 C++、C#、Java、Python 等多种语言，同时也支持 TCP/UDP 两种传输层协议和应用层的 HTTP 协议，其目的就是隐藏通信细节，并使导出的接口易于应用、更加通用，框架更加高性能，伸缩性更好。HP-Socket 具备跨平台能力，兼容 Windows 和 Linux 系统。HP-Socket 提供基于事件通知模型的回调式 API，因此 HP-Socket 也是一套异步通信框架。

HP-Socket 包括传输层组件类、SSL(Secure Sockets Layer，安全套接层)组件类和 HTTP/HTTPS 组件类的共计 29 个组件。其中传输层组件类共包含 12 个组件，涵盖 TCP 和 UDP 两种方式；高版本的 HP-Socket 的 TCP 组件全面支持 SSL。HP-Socket 的组件类中的各组件如表 21－1 至 21－3 所示。

表 21－1　传输层组件类

名称	组件接口 监听器接口	组件类	角色	协议	接收方式
TCP Server	ITcpServer ITcpServerListener	CTcpServer	Server	TCP	PUSH
TCP Pull Server	ITcpServer ITcpServerListener	CTcpPullServer	Server	TCP	PULL
TCP Pack Server	ITcpPackServer ITcpServerListener	CTcpPackServer	Server	TCP	PACK
UDP Server	IUdpServer IUdpServerListener	CUdpServer	Server	UDP	PUSH
UDP ARQ Server	IUdpArqServer IUdpServerListener	CUdpArqServer	Server	UDP	PUSH
TCP Agent	ITcpAgent ITcpServerListener	CTcpAgent	Client	TCP	PUSH
TCP Pull Agent	ITcpPullAgent ITcpAgentListener	CTcpPullAgent	Client	TCP	PULL

续表 21－1

名称	组件接口 监听器接口	组件类	角色	协议	接收方式
TCP Pack Agent	ITcpPackAgent ITcpServerListener	CTcpPackAgent	Client	TCP	PACK
TCP Client	ITcpClient ITcpClientListener	CTcpClient	Client	TCP	PUSH
TCP Pull Client	ITcpPullClient ITcpClientListener	CTcpPullClient	Client	TCP	PULL
TCP Pack Client	ITcpPackClient ITcpClientListener	CTcpPackClient	Client	TCP	PACK
UDP Client	IUdpClient IUcpClientListener	CUdpClient	Client	UDP	PUSH
UDP ARQ Client	IUdpArqClient IUdpClientListener	CUdpArqClient	Client	UDP	PUSH
UDP Cast	IUdpCast IUdpCastListener	CUdpCast	Client	UDP	PUSH

表 21－2　SSL 组件类

名称	组件接口 监听器接口	实现类	角色	接收方式
SSL Server	ITcpServer ITcpServerListener	CSSLServer	Server	PUSH
SSL Pull Server	ITcpPullServer ITcpServerListener	CSSLPullServer	Server	PULL
SSL Pack Server	ITcpPackServer ITcpServerListener	CSSLPackServer	Server	PACK
SSL Agent	ITcpAgent ITcpServerListener	CSSLAgent	Client	PUSH
SSL Pull Agent	ITcpPullAgent ITcpAgentListener	CSSLPullAgent	Client	PULL
SSL Pack Agent	ITcpPackAgent ITcpServerListener	CSSLPackAgent	Client	PACK
SSL Client	ITcpClient ITcpClientListener	CSSLClient	Client	PUSH
SSL Pull Client	ITcpPullClient ITcpClientListener	CSSLPullClient	Client	PULL
SSL Pack Client	ITcpPackClient ITcpClientListener	CSSLPackClient	Client	PACK

表 21－3　HTTP/HTTPS 组件类（表中内容来自 CSDN）

名称	组件接口 监听器接口	实现类	角色	基类
HTTP Server	IHttpServer IHttpServerListener	CHttpServer	Server	CTcpServer
HTTPs Server	IHttpServer IHttpServerListener	CHttpsServer	Server	CSSLServer
HTTP Agent	IHttpAgent IHttpAgentListener	CHttpsAgent	Client	CTcpAgent
HTTPs Agent	IHttpAgent IHttpAgentListener	CHttpsAgent	Client	CSSLAgent
HTTP Client	IHttpClient IHttpClientListener	CHttpClient	Client	CTcpClinet
HTTPs Client	IHttpClient IHttpClientListener	CHttpsClient	Client	CSSLClinet
HTTP Sync Client	IHttpSyncClient IHttpClientListener	CHttpSyncClient	Client	CTcpClinet
HTTPs Sync Client	IHttpSyncClient IHttpClientListener	CHttpsSyncClient	Client	CSSLClinet

按组件的角色分类，传输层组件类也分为 Client、Server 和 Agent 三种，其他两种组件类 SSL 和 HTTP 也是一样。

- **Client 组件**：TCP 客户端组件，定义了 Connect、Send、Recv 等事件响应接口。
- **Server 组件**：TCP 服务端组件，定义了 Listen、Accept、Send、Recv 等事件响应接口。
- **Agent 组件**：本质上是 TCP 客户端组件集群，可以代理多个 Client 请求。

其中，Server 组件基于 IOCP（Windows 下）模型和 epoll（Linux 下）模型，支持大规模的并发连接场景；Client 组件基于 select 和 poll 模型，比较适合小规模的并发场景。

从数据接收方式来分，可以分为 PUSH 组件、PULL 组件和 PACK 组件：

- **PUSH 组件**：当组件接收到数据后，触发监听器对象的 OnReceive 回调接口，该接口将数据以及数据长度回调给应用程序。
- **PULL 组件**：当组件接收到数据后，触发监听器对象的 OnReceive 回调接口，该接口只将接收缓冲区接收了多少数据告诉应用程序，应用程序还要调用 Fetch 接口去主动获取数据内容。
- **PACK 组件**：当组件接收到数据后，触发监听器对象的 OnReceive 回调接口，该接口以数据包为单位向应用程序回调。当然这要求数据在发送时就以数据包为单位，传输时在每个包前面加上 4 个字节的私有包头作为分隔符，这样一来组件在接收到包的时候才会区分出边界。





当 IClient 组件调用 Send、SendPackets、SendSmallFile 发送数据时，若发送链路比较繁忙，组件内部会缓存数据，等发送链路空闲时再行发送。

当 IServer 和 IAgent 组件调用 Send、SendPackets、SendSmallFile 发送数据时，可以通过 SetSendPolicy 设置不同的发送策略，不同的发送策略有不同的处理方式：

- **SP_PACK**：打包策略（默认），多个发送操作组合成一个操作，以便提高发送效率。
- **SP_SAFE**：安全策略，也是将多个操作组合成一个操作，同时避免发送缓冲区溢出。
- **SP_DIRECT**：直接策略，每一个发送操作直接投递，以便提高实时性。

当 IServer 和 IAgent 组件被回调 OnReceive 接口时，表示接收到了数据，但也有可能 OnClose 接口也在接收数据时被回调，我们可以通过 SetRecvPolicy 设置接收策略来避免这种尴尬局面：

- **RP_SERIAL**：串行策略（默认），对于单个连接，顺序回调 OnReceive 和 OnClose 接口以增强安全性。
- **RP_PARALLEL**：并行策略，对于单个连接，当同时收到数据和 Close 事件时，会在不同的线程中同时回调 OnReceive 和 OnClose 接口以提高并发性，但要求应用程序必须妥善处理调用 OnClose 接口带来的资源释放安全性问题。

我们以 TCP 方式为例来考察下 Server 和 Client 组件的工作流程，两种组件都遵循 TCP 通信的一般性特征。先来看 Server 组件的工作流程，如图 21－31 所示，具体如下：

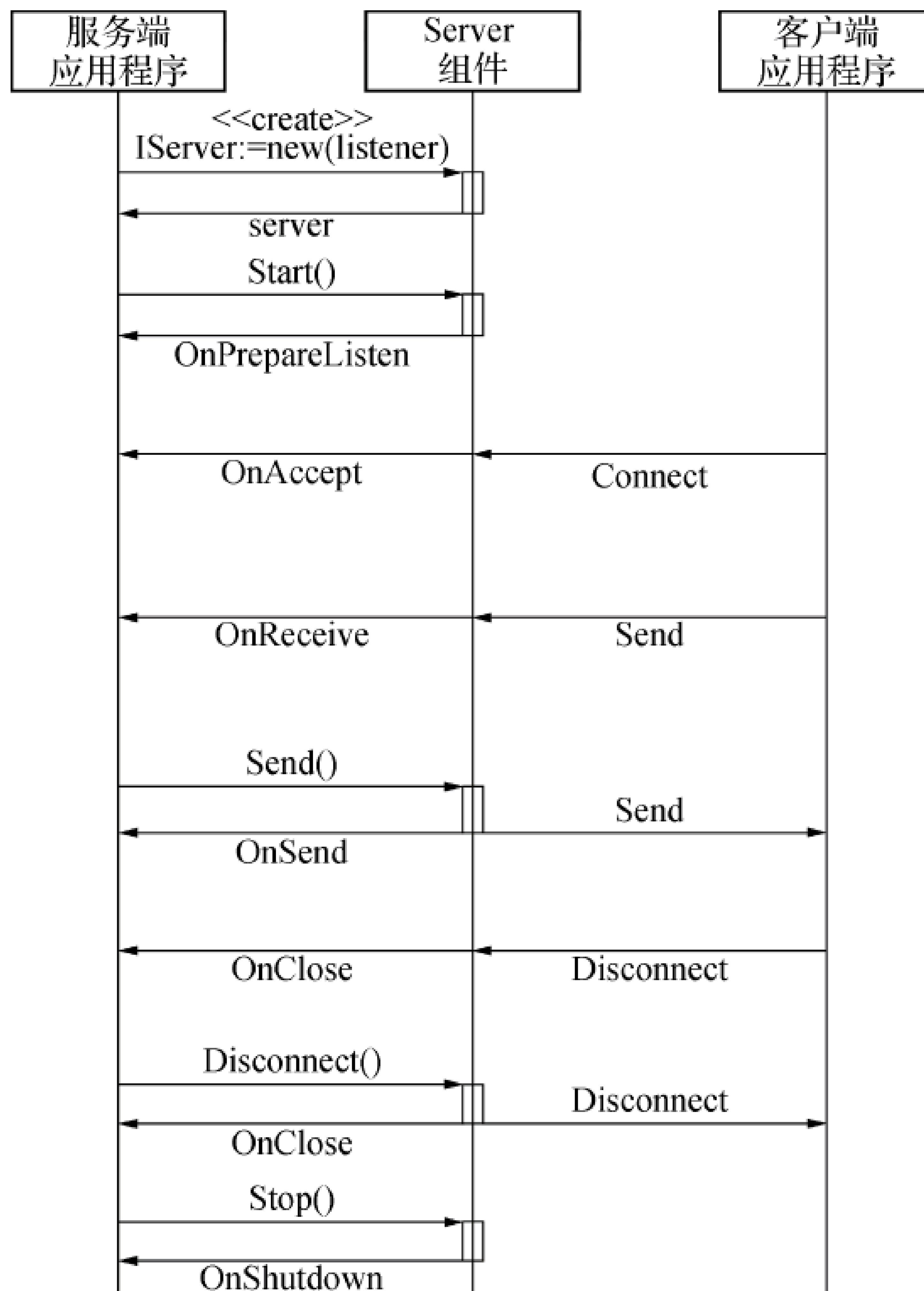


图 21－31　Server 组件的工作流程

（1）服务端应用程序调用 Start 方法以启动 Server 组件，如果调用成功会收到 OnPrepareListen 事件。

（2）客户端应用程序向服务端应用程序发起连接请求后，服务端应用程序将收到 OnAccept 事件。

（3）客户端应用程序向服务端应用程序发送数据时，服务端应用程序将收到 OnReceive 事件，并且根据 PUSH、PULL 和 PACK 接收方式分别对数据进行处理。

（4）服务端应用程序调用 Send 方法向客户端应用程序发出数据后，服务端应用程序将收到 OnSend 事件，表示发送完成。

（5）断开连接时，服务端应用程序将收到 OnClose 事件，表示连接断开。

（6）服务端应用程序调用 Stop 方法关闭 Server 组件，如果调用成功则收到 OnShutdown 事件，表示连接关闭。

Client 组件的工作流程如图 21－32 所示，具体如下：

（1）客户端应用程序调用 Start 方法向服务端应用程序发起连接请求，如果连接成功则会先后接收到 OnPrepareConnect 和 OnConnect 事件，分别表示开始建立连接和连接建立成功。

（2）客户端应用程序调用 Send 方法向服务端应用程序发出数据后，客户端应用程序将收到 OnSend 事件，表示数据发送完毕。

（3）服务端应用程序向客户端应用程序发送数据时，客户端应用程序将收到 OnReceive 事件，并根据 PUSH、PULL 和 PACK 接收方式分别对数据进行处理。

（4）断开连接时，客户端应用程序将收到 OnClose 事件，表示连接断开。

（5）客户端应用程序调用 Stop 方法关闭 Client 组件，如果调用成功则再次收到 OnClose 事件，表示连接关闭。

Agent 组件是多个 Client 组件的代理，因此具有 Client 的大部分特性。Agent 组件的工作流程如图 21－33 所示，各个步骤与 Client 组件的极为相似，其代表的含义也基本一致：

（1）客户端应用程序调用 Start 方法启动 Agent 组件，如果调用成功则返回 True。

（2）客户端应用程序调用 Connect 方法向服务端应用程序发起连接请求，如果连接成功则会先后接收到 OnPrepareConnect 和 OnConnect 事件。

（3）客户端应用程序调用 Send 方法向服务端应用程序发出数据后，客户端应用程序将收到 OnSend 事件。

（4）服务端应用程序向客户端应用程序发送数据时，客户端应用程序将收到 OnReceive 事件。

（5）断开连接时，客户端应用程序将收到 OnClose 事件。

（6）客户端应用程序调用 Stop 方法关闭 Agent 组件，如果调用成功则收到 OnShutdown 事件。

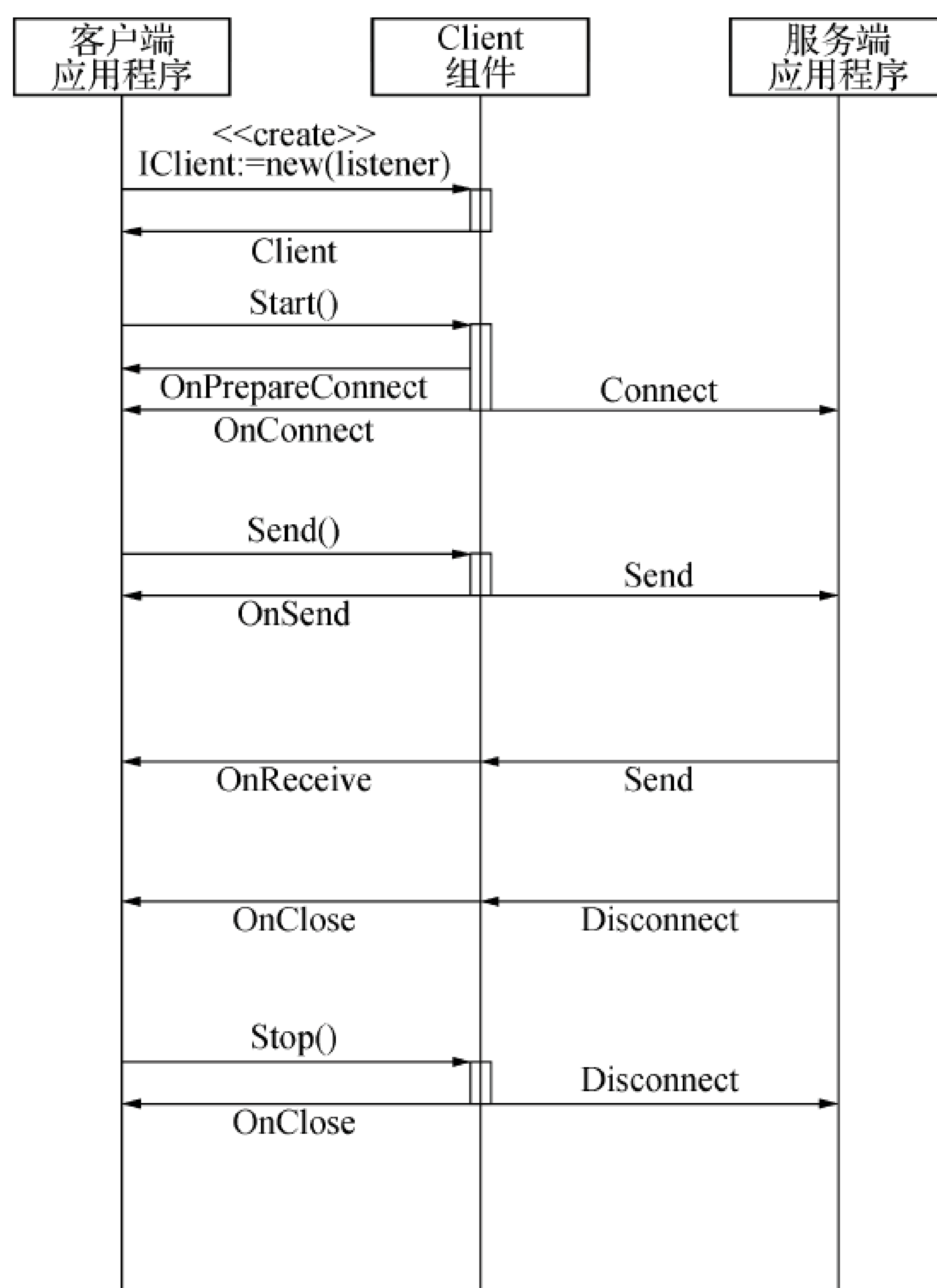


图 21－32　Client 组件的工作流程

21.3　消息队列

消息队列(Message Queue,MQ)就像存放消息的容器管道,一端(生产者)负责向管道内push 消息,另一端(消费者)则负责从管道内 pull 消息。在管道的中间有一个有限的缓冲区,负责一定时间段内的持久化工作,以便于执行一些“削峰填谷”的操作。基于此,MQ 在一定程度上定义了生产者与消费者之间的工作界面。

常用的 MQ 包括 Kafka、ActiveMQ、RabbitMQ、RocketMQ、ZeroMQ 等。不过一般不认为ZeroMQ(ZMQ)是消息队列,因为它是对 socket 和 IPC 机制的一种封装,更类似于 ACE 这样的通信框架,只是内部的实现逻辑借用了消息队列的一些思想,因此通常会将 ZMQ 看作通信框架库。表 21－4 列出了常用消息队列。

在这里我们也简单列出 ActiveMQ、RabbitMQ、RocketMQ 和 Kafka 这几种典型消息队列的特点,如表 21－5 所示。本节我们不准备介绍所有的消息队列,只是浅显地考察下 ZMQ和 RocketMQ。因为在 MQ 的家族中,各种消息队列的大多数特性都是大同小异的,其主要的共同特性(例如持久化、订阅机制等)的实现思想也基本一致,并没有特别革命性的突破。

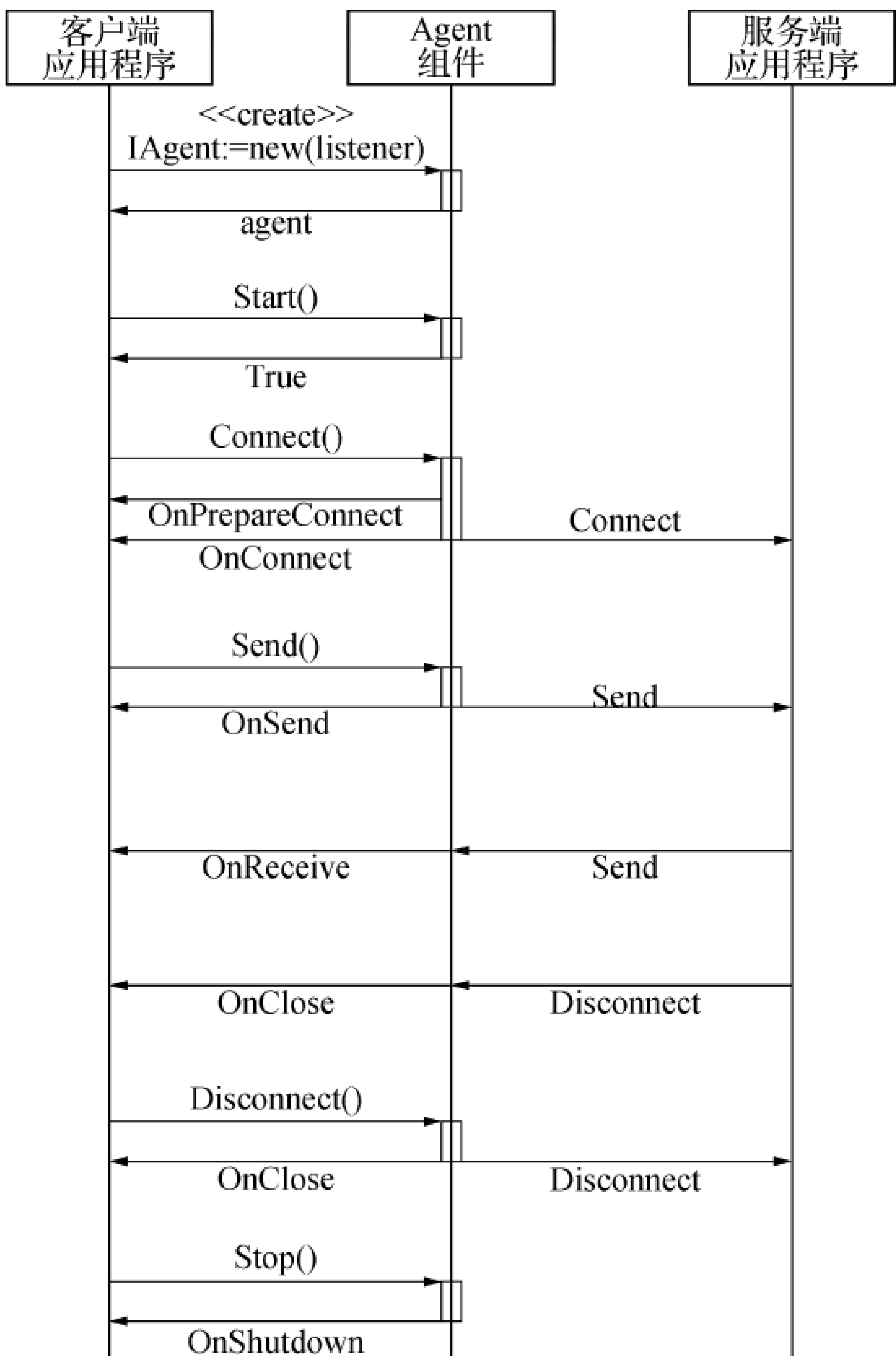


图 21－33　Agent 组件的工作流程

表 21－4　常用消息队列

消息队列	开发者	开发语言	队列说明	是否跨平台
Kafka	Apache 软件基金会开发的一个开源流处理平台	使用 Scala、Java 开发，支持 C、C++、Erlang、Java、. NET、Perl、PHP、Python、Ruby、Go、JavaScript 等语言调用	分布式发布-订阅消息系统	支持 JVM 跨平台特性
ActiveMQ	Apache 软件基金会开发的一个开源消息中间件	使用 Java 语言开发，支持 Java、C、C++、C #、Python、Ruby、Perl 等语言调用	支持 JMS1. 1 和 J2EE 1. 4 规范的开源消息总线	支持 JVM 跨平台特性
RabbitMQ	Rabbit 公司开发的开源消息中间件	使用 Erlang 语言开发，支持 Python、Java、Ruby、PHP、C #、JavaScript、Go、Elixir、Objective-C、Swift 等语言调用	高级消息队列协议（AMQP）的开源消息代理软件	支持 Windows、Linux 和 Mac OS 等平台
MetaMQ	淘宝开源的分布式消息中间件	使用 Java 语言开发，且只支持 Java 语言调用	高可用分布式消息中间件	支持 JVM 跨平台特性

续表 21－4

消息队列	开发者	开发语言	队列说明	是否跨平台
MSMQ	微软闭源的消息队列	使用.NET 语言开发，支持.NET 语言调用	在多个不同的应用之间实现相互通信的一种异步传输模式	支持.NET 虚拟机跨平台特性
PhxQueue	微信开源的一款基于 Paxos 协议实现的消息队列	使用 C++ 语言开发，支持 C++ 语言调用	高可用性、高吞吐量和高可靠性的分布式消息队列	支持 Linux

表 21－5　几种典型消息队列的特点

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
生产者–消费者模式	支持	支持	支持	支持
发布–订阅模式	支持	支持	支持	支持
请求–应答模式	支持	支持		
API 完备性	高	高	高	高
多语言支持	支持，Java 优先	语言无关	只支持 Java	支持，Java 优先
单机吞吐量	万级	万级	万级	十万级
消息延迟		微秒级	毫秒级	毫秒级
可用性	高(主从)	高(主从)	非常高(分布式)	非常高(分布式)
消息丢失	低	低	理论上不会丢失	理论上不会丢失
消息重复		可控制		理论会有重复
文档的完备性	高	高	高	高
提供快速入门	有	有	有	有
首次部署难度		低		中
社区活跃度	高	高	中	高
商业支持	无	无	阿里云	无
成熟度	成熟	成熟	比较成熟	成熟
特点	功能齐全，被大量开源项目使用，生态成熟，兼容性好	由于 Erlang 语言的并发能力，性能很好，支持 AMQP 协议	各个环节分布式扩展设计，主从 HA；支持上万个队列；多种消费模式，性能很好，模型简单易用	
支持协议	OpenWire、STOMP、REST、XMPP、AMQP	AMQP	自己定义的一套(社区提供 IMS，但不成熟)	
持久化	内存、文件、数据库	内存、文件	磁盘文件	
事务	支持	不支持	支持	
负载均衡	支持	支持	支持	

续表 21－5

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
管理界面	一般	好	有 Web console 实现	
部署方式	独立、嵌入	独立	独立	

21.3.1　ZeroMQ

ZeroMQ(ZMQ)是一种基于消息队列的非持久性网络库(类似于 socket 库),支持多线程,具有较低的开发门槛。ZMQ 是基于 C 语言开发的,可以绑定 C、C++、Java、.NET、Python 等 30 多种开发语言,并且支持 Linux、Windows、Mac OS X 等平台。ZMQ 在 OSI 参考模型中位于应用层之下、传输层之上。虽然 ZMQ 的名称中带有“MQ”,但一般不将它归为消息队列,而更多地将它看作通信框架组件库。ZMQ 具有以下特点:

- 采用了去中心化的设计思想。
- 生产者与消费者内部采用 TCP 方式传输消息。
- 支持处理连接中断后再次重连等网络异常处理机制。
- 以消息(MSG)而非字节或流为单位收发数据,因此不存在类似 TCP 粘包这样的问题。
- 通过 SENDMORE/RECVMORE 方法支持对大数据量的分包收发机制。
- 任意时刻消息只被一个线程持有,因此不需要多线程锁,实现了去锁化。
- 支持通过水位标来控制 I/O 流量。
- 兼容支持进程内通信、进程间通信(含跨主机)和广播等多种通信组网环境。
- 支持异步 I/O 操作。
- 支持并行运行,支持分布式部署。
- 除了网络通信,ZMQ 也封装了消息队列和线程调度等机制。
- 与其他消息队列(RabbitMQ、ActiveMQ 和 MSMQ 等)相比 I/O 效率最高。

当然,ZMQ 也具有其他通信框架库固有的特点,虽然在高并发下本身一般不会出现稳定性问题,但有可能使本地缓存被填满而导致消息丢弃,且缺乏类似 TCP 的丢包重传机制,因此不具备其他消息队列所固有的持久化能力。

如图 21－34 所示,在 ZMQ 框架中,每个 I/O 线程都绑定了一个轮询器(Poller)。Poller 是采用 Reactor(反应堆)模型实现的,通过 select、Kequeue 或 epoll 机制实现了 I/O 多路复用和完成事件通知。I/O 线程通过注册的方式与 Poller 绑定,Poller 会将完成事件回调给 I/O 线程。

在 ZMQ 框架下,I/O 线程通过 Mailbox 来与主线程(也具有 Mailbox 数据结构)进行消息命令的交换,一个 Mailbox 就像专用的管道承载收发命令。主线程与 I/O 线程通过管道机制进行消息报文的交换,这些消息报文就是 I/O 的数据。从主线程向 I/O 线程传输时采用 Outbound Pipe,反之则采用 Inbound Pipe,且二者均支持 write/read 操作,分别用于发送和接收消息数据。



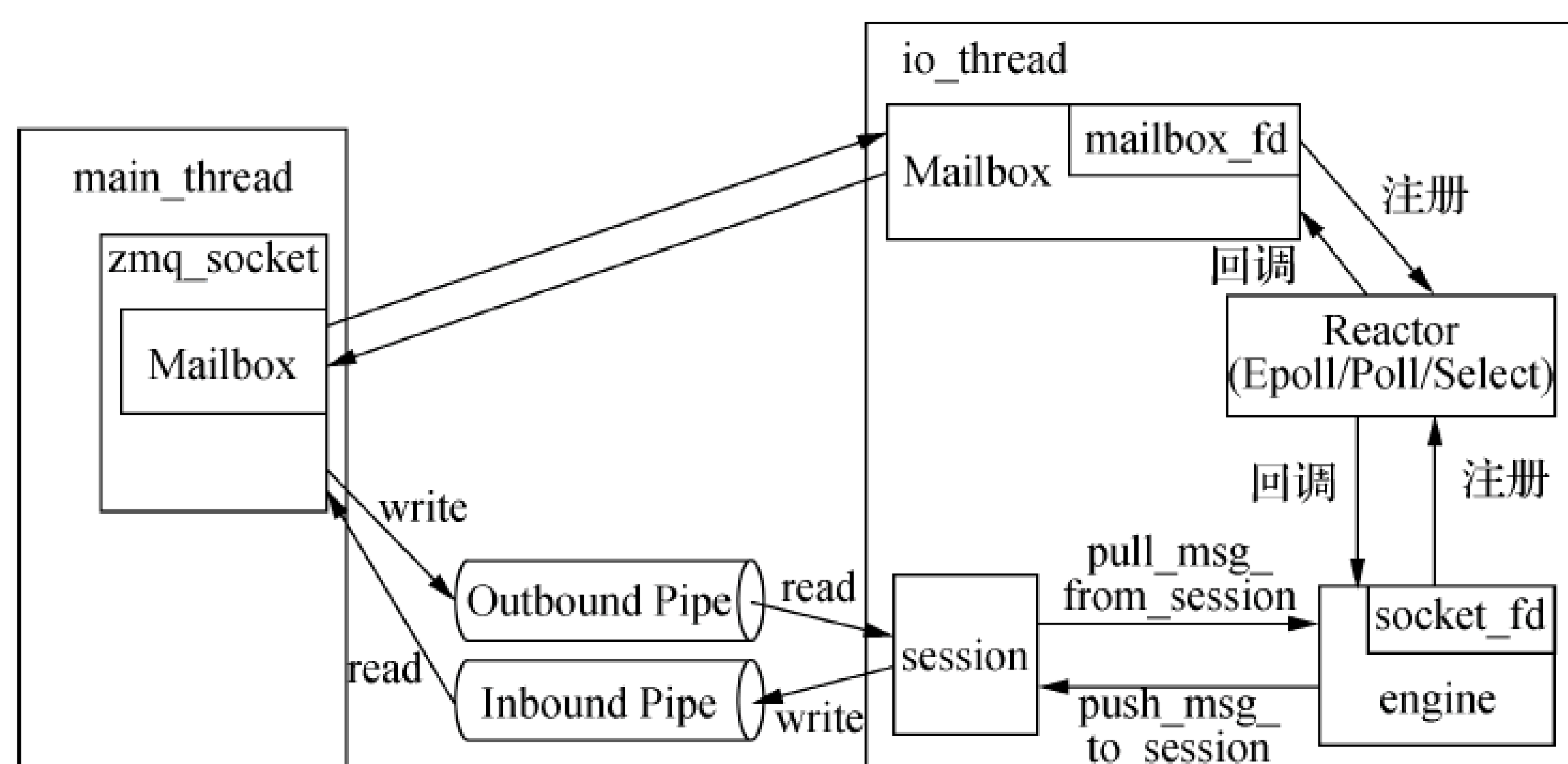


图 21－34　ZMQ 架构视图

ZMQ 支持发布–订阅模型、请求–应答模型和管道模型三种通信工作模型。

1）发布–订阅模型

消息的接收者被定义为“订阅端”，发布者被定义为“发布端”。订阅端需要向发布端订阅消息接收请求。在订阅成功之前，发布端发出的消息都会被丢弃。订阅端只负责接收消息而不能反馈消息，如果需要反馈则应采用额外的链路来实现。

如图 21－35 所示，在这个模型中，订阅关系可以是一对多，即单个订阅者可以向多个发布者订阅，或者多个订阅者向单个发布者订阅。并且为了保证公平，单个订阅者可以平均地从多个发布者读取消息。当然，这个规则也可以被配置和改变，在 ZMQ 3.0 以前的版本中，订阅者制定这些规则，而在 ZMQ 3.0 及以上的版本中由发布者制定规则。

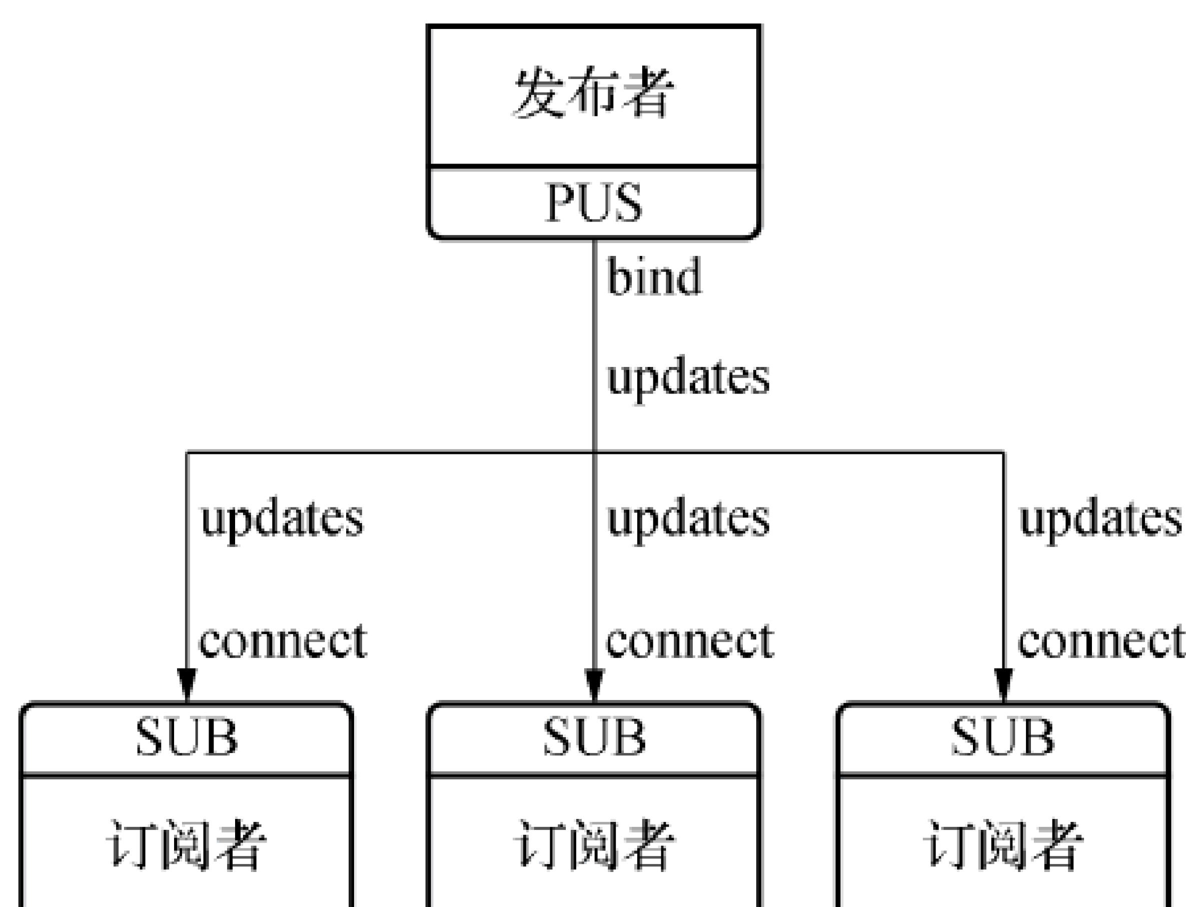


图 21－35　发布–订阅模型

由于 ZMQ 是采用 TCP 方式传送数据的，在订阅者和发布者启动的过程中彼此要建立 TCP 连接，因此会耗费一定的握手时间，在这个过程中订阅者可能会丢失若干发布者发布的消息。

2）请求–应答模型

这种模型也被称为一问一答模型，即请求与应答必须是一对一的。可以将这种模型理解为同步的，消息的发送者只有收到了请求才会发出回复。如果发出了请求没有收到回复，

再次发出请求消息时 ZMQ 库会抛异常。

如图 21－36 所示，在这个模型中，请求和应答是由元封包表示的，每个元封包包括了若干个帧。例如图 21－37 中的两个元封包就是含有识别码和不含识别码的两种形态的包。在含有识别码的元封包中，第一帧用于存放消息接收端（客户端）的身份识别码，这是由调用者指定的，ZMQ 内部会维护一个以该识别码为键、客户端地址为值的 MAP 结构；第二帧作为分隔符一般是空的；第三帧则是需要发送的数据。其实在不含识别码的元封包中也有隐藏的识别码，只是在元封包中不体现，且这个识别码是由 ZMQ 默认自动生成的。

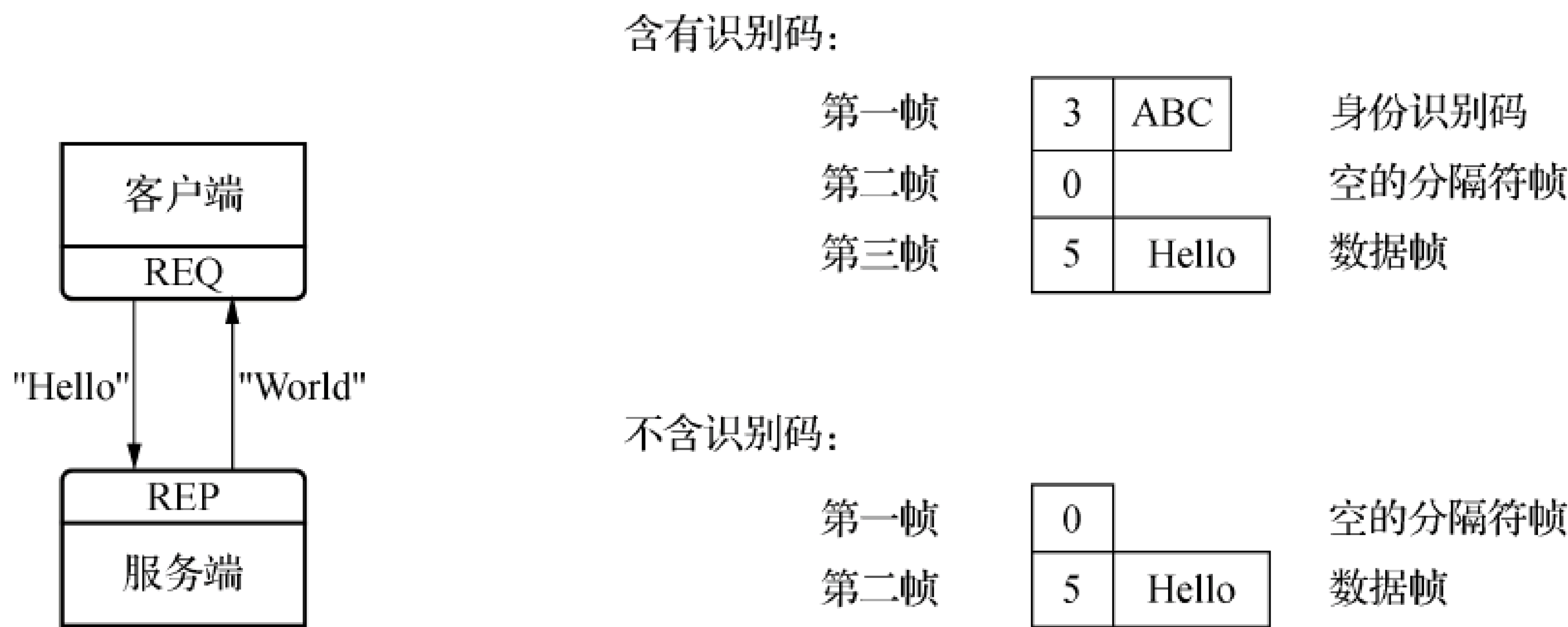


图 21－36　请求–应答模型　　**图 21－37　元封包的两种形态**

在 ZMQ 中，当遇到客户端与服务端是一对多或者多对一的关系时，作为请求者的客户端每次只跟一个应答者交流，如果请求者连接了多个应答者，那么请求会同时分发到每一个应答者。作为应答者的服务端每次也只是与一个请求者交流，如果应答者连接了多个请求者，则会以平均的方式从各个请求者读取请求，但最先响应的是最后读取的请求。

3）管道模型

在一个管道中，消息的接收端被定义为“Pull 端”，发布端被定义为“Push 端”，二者的关系可以是一对多或多对一，这样有利于以并行的方式解决 Pull 端读取能力不足或 Push 端写入能力不足的情况。

如图 21－38 所示，在这个模型中，管道是单向的，消息只能由 Push 端发往 Pull 端。在一对多场景下，Push 端采用了平均主义的负载均衡机制，所有消息均衡地发布到每个 Pull 端。管道模型内部采用了 LRU（Least Recently Used，最近最小使用）的算法来找到最近最久未工作的闲置 Worker 线程来承担消息的运输工作。

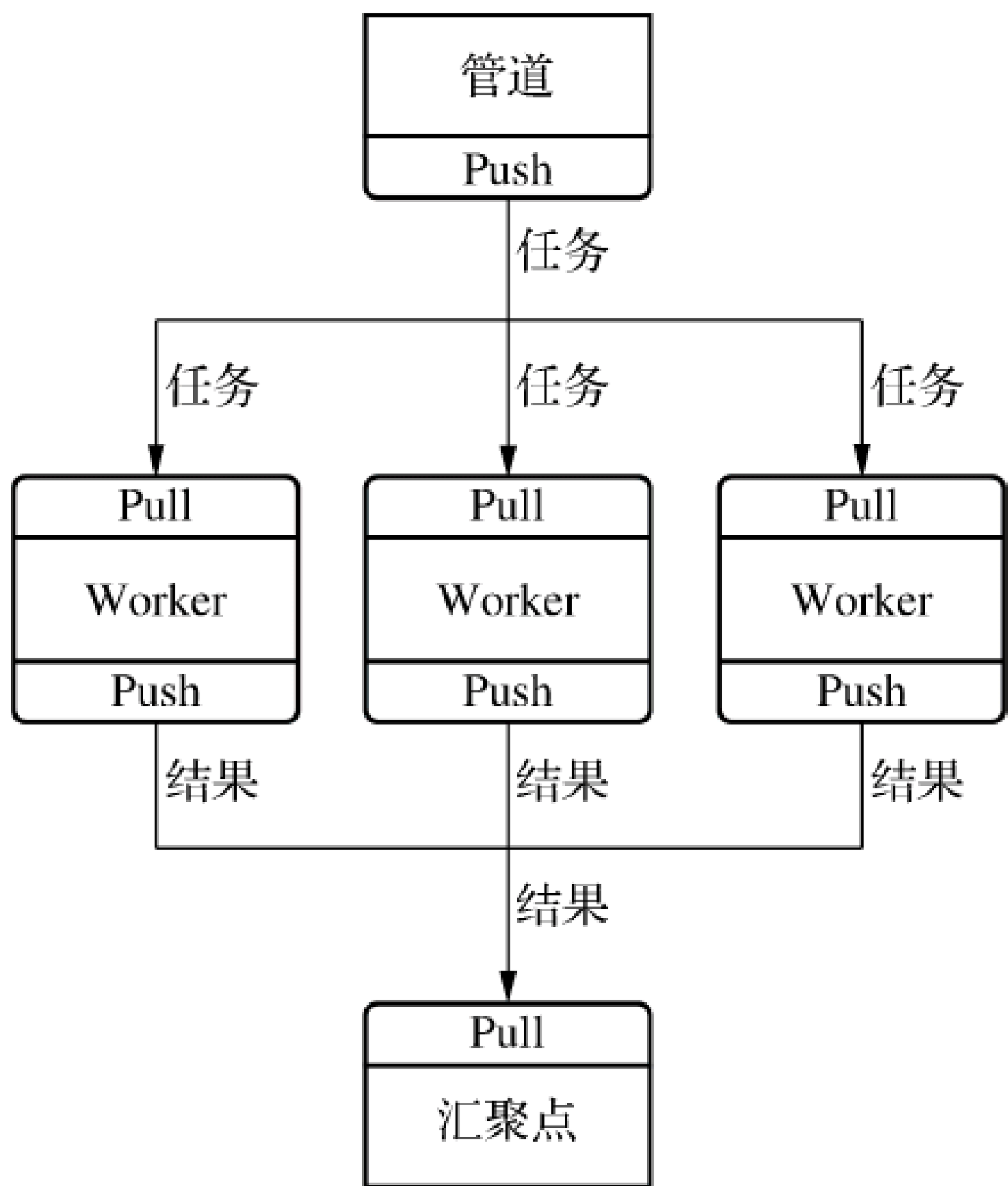


图 21－38　管道模型

在管道模型中，管道具有很有限的持久化能力，在没有 Pull 端的情况下发布的消息不会立即

丢弃,但也不会无限保留。

从上述三个模型中我们也可以看出,ZMQ 会区分消息的拥有方和索取方,我们将它们分别定义为服务端和客户端,二者没有明确的启动顺序,但只有二者都启动了,工作模型才会起作用。无论是服务端还是客户端都具有主线程和 I/O 工作线程,其启动和工作的流程如图 21-39 所示。

客户端
主线程
I/O线程
调用zmp_init创建context指针
创建I/O线程对象
创建Mailbox
启动I/O线程
调用zmp_socket根据指定的pattern创建socket指针
调用zmp_connect根据指定的通信方式连接server
创建session_t指针
绑定session指针到I/O线程session与主线程之间通过管道通信
创建zmp_connecter_t指针
绑定connecter
调用zmp_send将msg写入session关联的管道
session从管道中读msg
调用zmq_close,关闭socket
调用zmq_term,销毁context

调用start_connecting函数连接服务端
连接服务端成功后创建zmp_init_t指针
发送peer_identity
回复收到peer_identity
engine将msg发送给server

服务端
I/O线程
主线程
调用zmp_init创建context指针
创建I/O线程对象
创建Mailbox
启动I/O线程
调用zmp_socket根据指定的pattern创建socket指针
调用zmp_bind根据指定的通信方式监听客户端的连接请求
创建zmp_listener_t指针
绑定listener
接受客户端连接请求,创建zmp_init_t指针
创建session_t指针
I/O线程session与主线程之间通过管道通信
engine从client接收msg,session将msg写入管道
主线程调用zmp_recv从session关联管道读取msg
调用zmq_close,关闭socket
调用zmq_term,销毁context

图 21-39 ZMQ 的工作流程

在这个工作流程中有以下几点注意事项：

- 主线程启动 I/O 线程，并为 I/O 线程创建 Mailbox 数据结构。
- ZMQ 的工作机制与 socket 中 TCP 的机制类似，例如 connect、bind、listen 等操作，并且客户端的主线程要与服务端的主线程建立连接，以实现命令交互。
- 每一个 I/O 线程都会创建 session，session 通过管道与主线程通信。
- 客户端与服务端之间的 I/O 工作线程负责消息的实际交互，这是由 I/O 线程的 engine 实现的。
- 客户端与服务端之间的 I/O 线程交互消息的机制遵照上述三种模型。

21.3.2　RocketMQ

RocketMQ 是阿里发布的消息中间件，是一种常用的典型消息队列，具有开源、低延迟、高可靠性、高可伸缩性、使用门槛低、支持分布式和高可用部署等特点，具体如下：

- 支持发布-订阅和点对点（P2P）两种消息模型。
- RocketMQ 中的所有消息都是持久化的，可以落盘到文件中存储。
  - RocketMQ 先将消息写入页面高速缓存，然后刷盘（写入磁盘），这样能保证内存与磁盘中都有一份相同的数据，访问时直接从内存读取即可。
  - 同时，RocketMQ 对文件读写做了优化，采用内存映射方式实现读写。这样就把磁盘文件映射到了内存地址空间里，避免了内核态空间到用户态空间的拷贝。
- 正是由于具有强大的持久化能力，消息的优先级机制的成本会很高，因此 RocketMQ 支持以其他方式变相或者部分支持消息优先级机制。例如支持单独配置某个高优先级消息队列，并与其他普通优先级队列区别开。
- 具有强大的消息堆积能力，支持 10 亿级别的消息堆积，不会因为消息堆积而影响性能。
- 支持先进先出（FIFO）机制，可以支持严格的消息顺序传递策略（有序消息类型）。
- 支持 Pull 和 Push 两种消息处理模式。
- Broker 端支持以多个过滤进程的方式对消息进行过滤（Message Filter）。
- 单一队列具有百万级的消息堆积能力。
- 广泛支持 JMS、MQTT 等多种协议。
- 提供 Docker 镜像用于隔离测试和云集群部署。
- 具有友好的运维界面，提供配置、指标和监控等功能丰富的 Dashboard（仪表盘）。
- 服务端使用 Java 语言编写，客户端则支持 Java 和 C++ 语言。

与大多数消息队列一样，RocketMQ 也定义了生产者（Producer）和消费者（Consumer）两种角色。在发布-订阅模型中，Consumer 订阅了 Broker（一种缓存代理，相当于生产者和消费者之间的缓存中介）上的某个 Topic（是一种对消息的逻辑分类，相当于主题消息库的概念，RocketMQ 中的各组件都是围绕着 Topic 建立起对应关系的），当 Producer 发布消息到 Broker

上时，Consumer 就可以收到这些发布的消息，如图 21－40 所示。

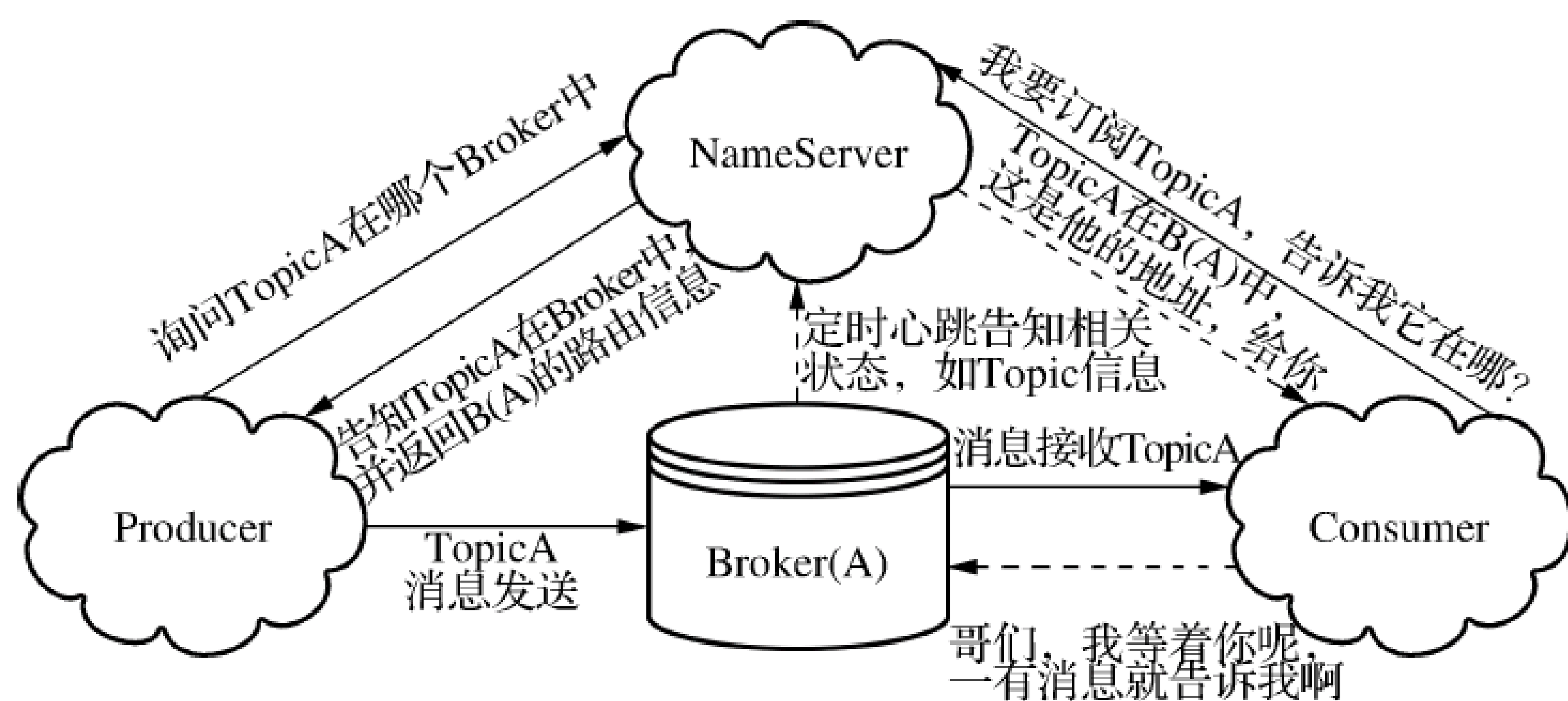


图 21－40　RocketMQ 的工作流程（图片来自 CSDN）

如图 21－41 所示，一个 Topic 可以分布在各个 Broker 上，我们将一个 Topic 分布在一个 Broker 上的子集定义为一个 Topic 分片，分片的目的就是突破单点的资源（带宽、CPU、内存或文件存储等）限制以实现水平扩展。将 Topic 分片再切分为若干等份，其中的一份就是一个 Queue。每个 Topic 分片等分的 Queue 的数量可以不同，由用户在创建 Topic 时指定。Consumer 根据 Broker 分配的 Queue 来消费数据。Queue 是负载均衡过程中资源分配的基本单元。

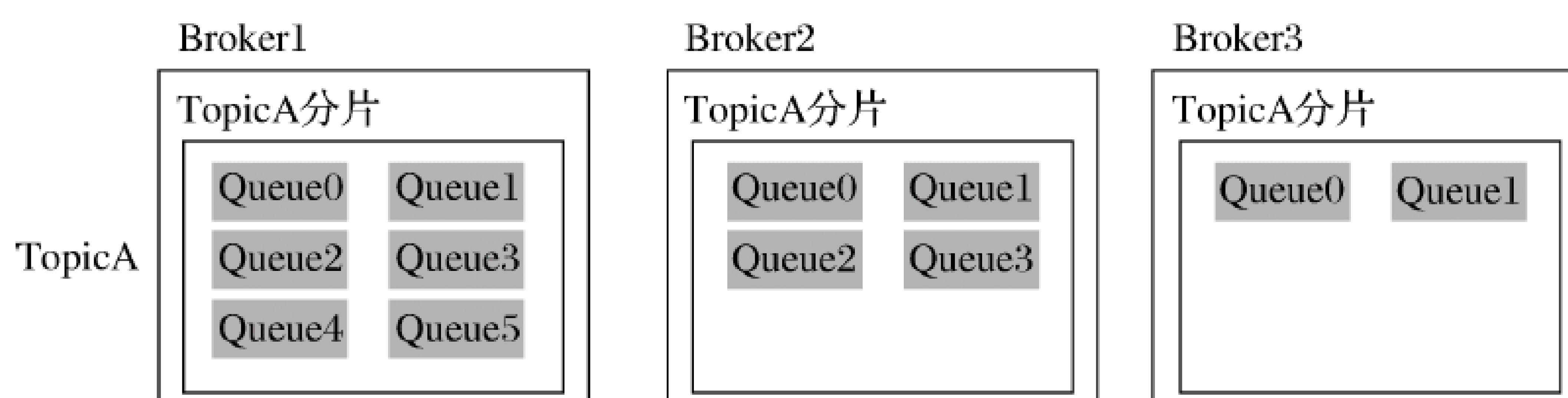


图 21－41　在各个 Broker 上分布的 Topic 分片与 Queue

在一个 Consumer Group 内，Queue 与 Consumer 之间的对应关系是多对一关系，即一个 Queue 最多只能分配给一个 Consumer，一个 Cosumer 可以被分配到多个 Queue。由于每个 Queue 只有一个 Consumer，从而可以避免消费过程中的多线程处理切换和资源锁定，有效提高各 Consumer 消费的并行度和处理效率。

1）RocketMQ 的生产者角色

RocketMQ 具有同步发送（请求和响应一来一回）、异步发送（Producer 通过回调接口接收来自 Broker 的响应通知）、单向发送（Producer 只负责生产和发送消息而不管 Broker 有没有发回响应通知或触发回调）和死信队列（Dead Letter Queue）4 种发送模式，其中死信队列用于存放由于某些原因而无法传递的消息，例如处理失败的消息或已经过期的消息等。

2）RocketMQ 的消费者角色

RocketMQ 消费者角色具有集群消费、广播消费和基于集群消费的模拟广播消费三种消费者模型。

(1) 集群消费

如图 21 - 42 所示,在这种模型中,每条消息只会被 Consumer 集群中的任意一个成员消费一次,类似于"单播",每条消息不会被拷贝分发。每个成员订阅的 Topic 的 Tag 都一样,它们的消费进度都存储在 Broker 上。这种模型并不能保证每一次消息失败时的重试都被精准投递到同一个 Consumer 成员中,当然 RocketMQ 也不允许无限制地失败重试。

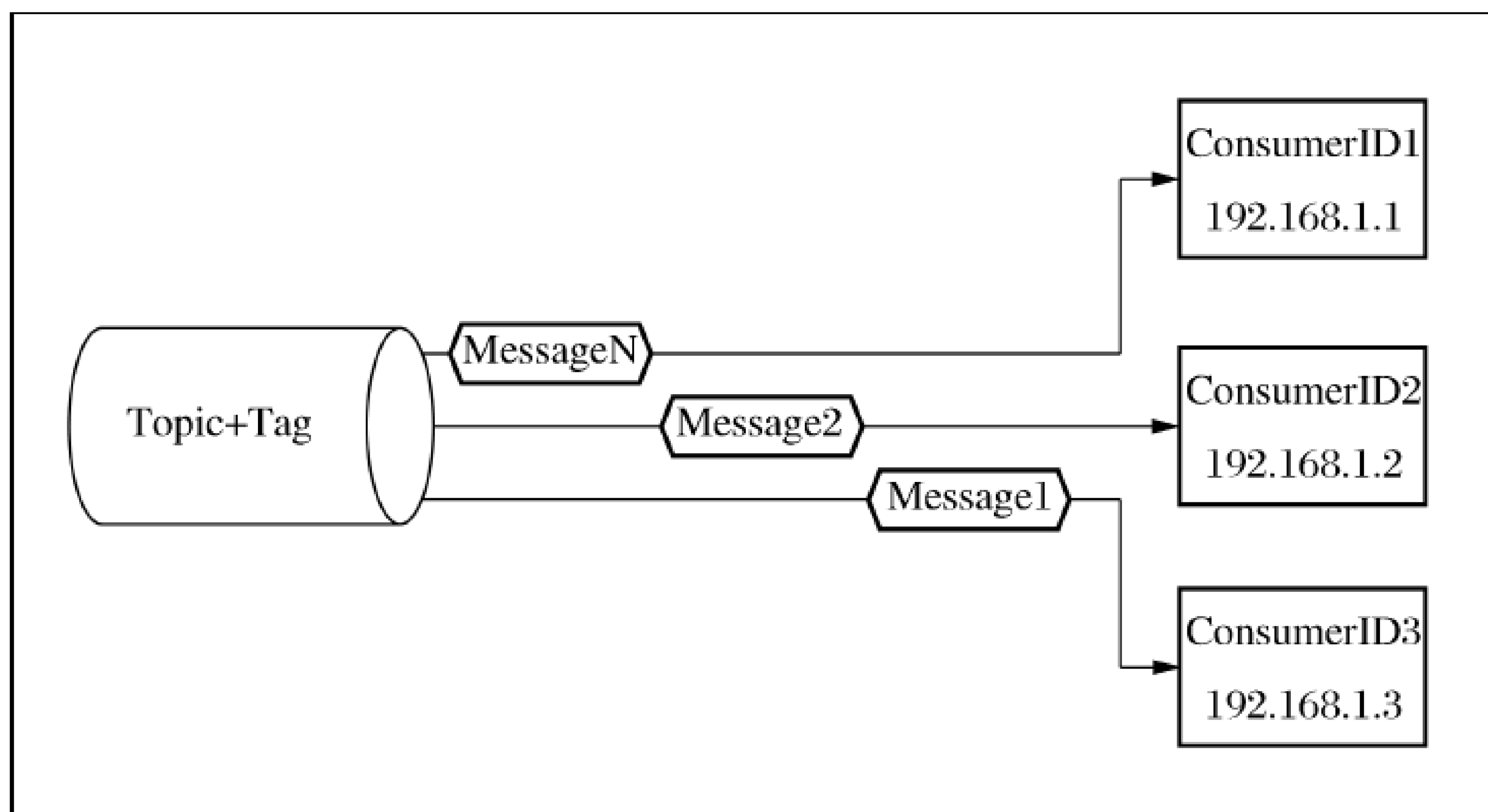


图 21 - 42　集群消费的模型

(2) 广播消费

如图 21 - 43 所示,在这种模型中,每条消息会被 Consumer 集群中的每一个成员消费一次,有几个成员就消费几次,类似于"广播",每条消息都会被拷贝分发。每个成员订阅的 Topic 的 Tag 都一样,而且这种模型中 Consumer 的消费进度存放在 Consumer 自己身上而不是 Broker 中,因此 Broker 不掌握全局进度,这可能会导致消息的重复。如果模型中消息的消费失败了,Broker 是不会进行重试投递的,因为 Broker 不知道消费失败了。

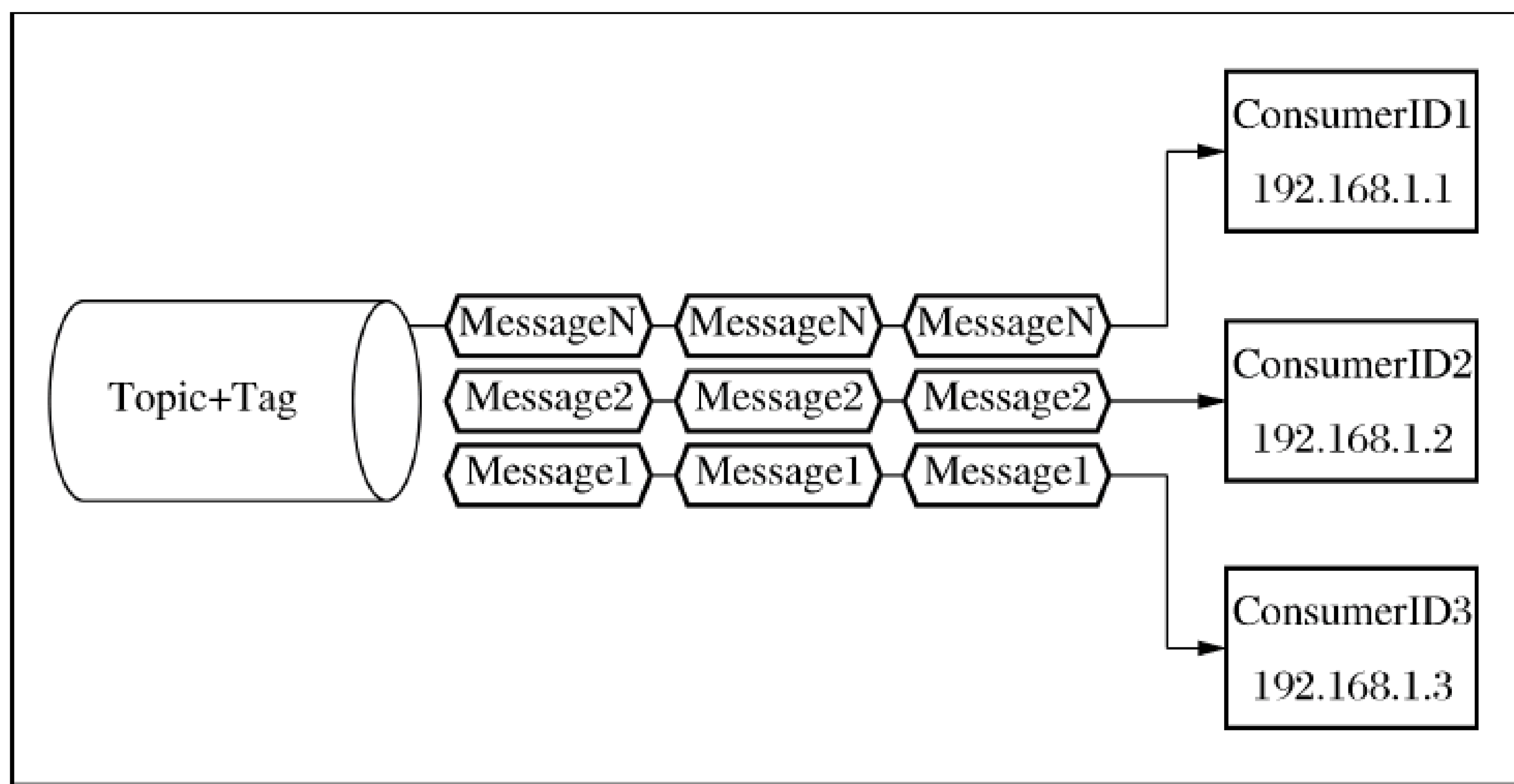


图 21 - 43　广播消费的模型

(3) 基于集群消费的模拟广播消费

如图 21 - 44 所示,在这种模型中,每个 Consumer 都属于不同的 Consumer 集群,当然每个 Consumer 集群也可以包含多个 Consumer,每个 Consumer 的消费逻辑可以不一样。

在这种模型中，每个集群都订阅相同的 Topic。针对每一条消息，Broker 会对每个集群都做拷贝分发，因此也实现了广播的效果。

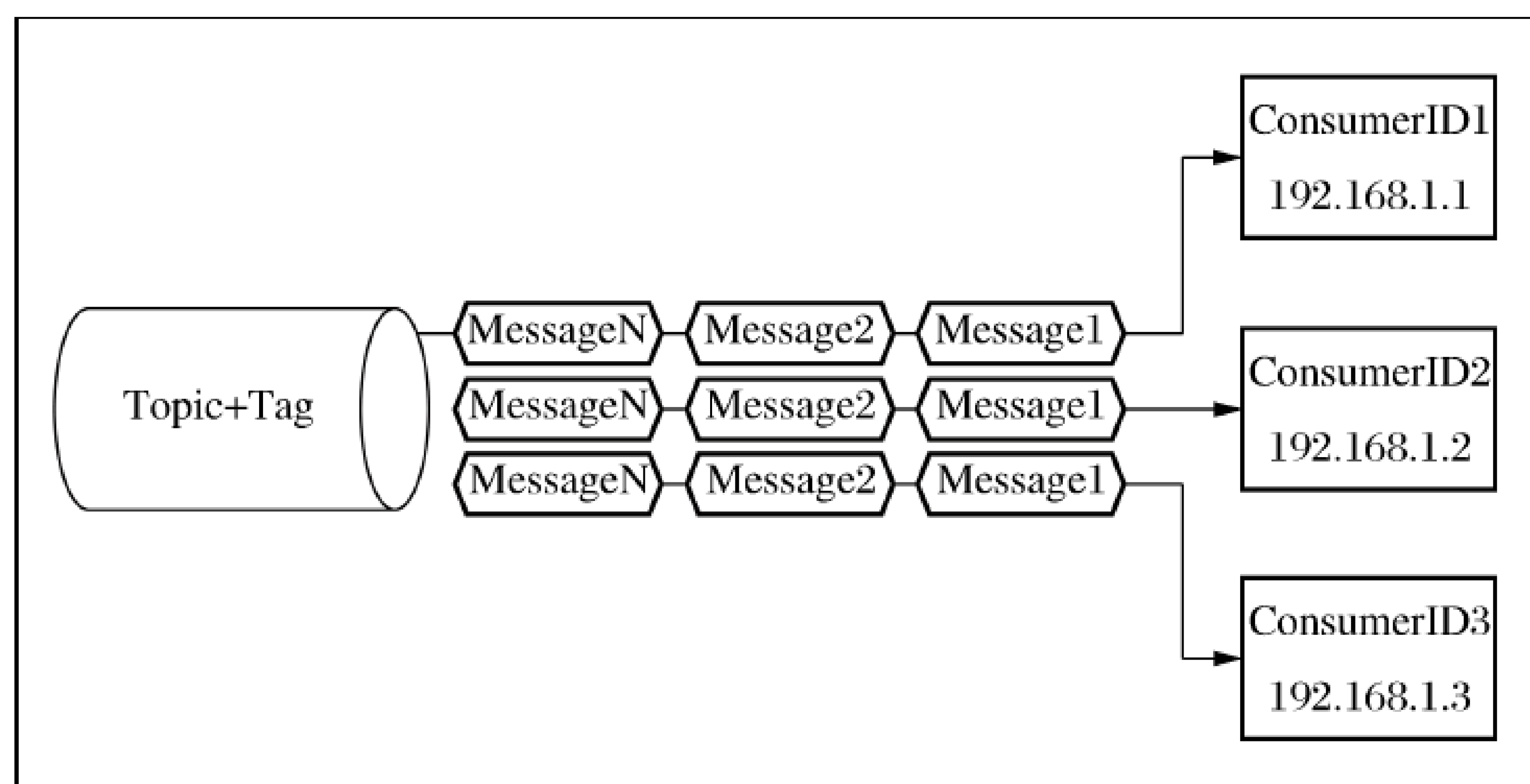


图 21－44　基于集群消费模拟广播消费的模型

在上述几个模型中，消费端的实现逻辑必须要保证消息的幂等性，即无论消息被处理多少次，最终结果都一样。因为 RocketMQ 虽然支持消息顺序传递，但并没有过多关注顺序消费和重复消费的问题。

3）RocketMQ 的消息类型

RocketMQ 的消息分为普通消息、有序消息和延时消息三大类，其中普通消息可以理解为无序消息，而有序消息又分为两小类，即全局有序消息和局部有序消息，如图 21－45 所示。

- **全局有序消息**：一个 Broker 对每个 Topic 只建有一个消息传送队列，吞吐量比较小。
- **局部有序消息**：一个 Broker 对每个 Topic 可以建有多个消息传送队列，每个队列要保证彼此的消息不重复。

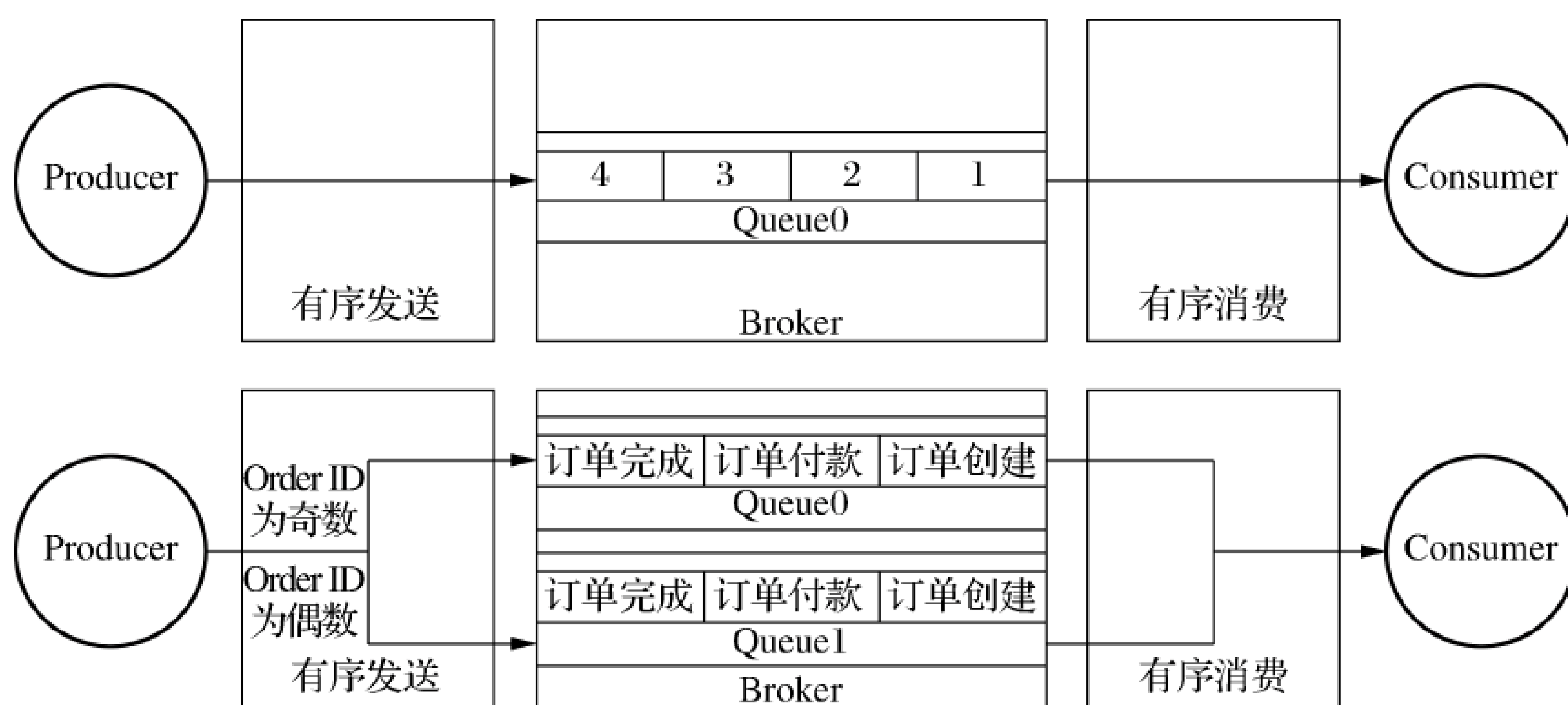


图 21－45　全局有序和局部有序消息

4）RocketMQ 的框架

RocketMQ 整体上分为 4 个部分：生产者集群（Producer Cluster）、消费者集群（Consumer

Cluster)、缓冲区集群(Broker Cluster)和名称服务集群(NameServer Cluster),如图 21 - 46 所示。

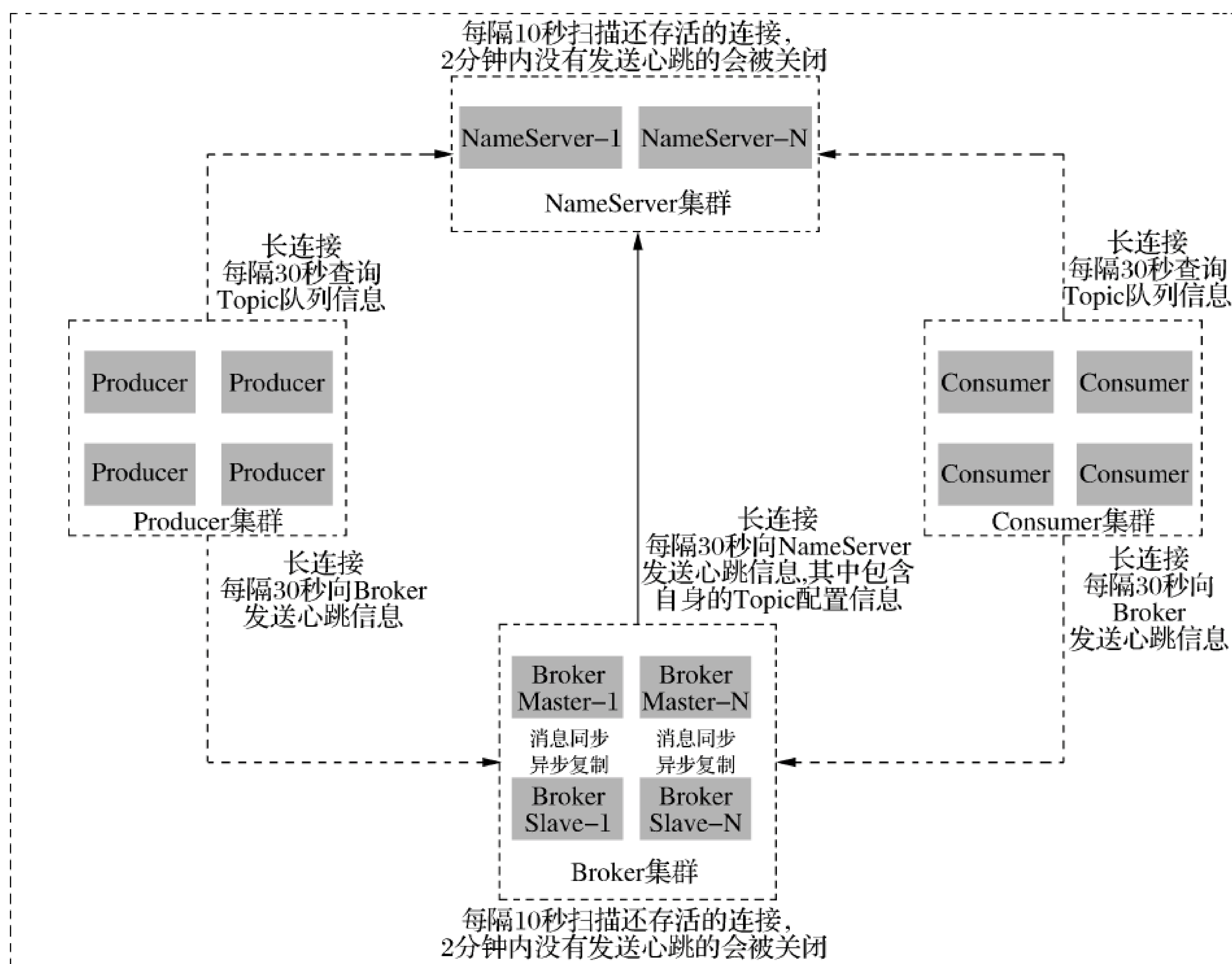


图 21 - 46　RocketMQ 架构

➢ **生产者集群(Producer Cluster)**:负责生产消息,集群中的每个 Producer 成员都与 NameServer Cluster 中的一个节点维持长连接关系。
  - 每个 Producer 成员定期从 NameServer 获取 Topic 的路由信息,并与提供 Topic 服务的 Broker 建立长连接,同时维持保活心跳。
  - Producer 只将消息发送到 Broker 上而非直接发给 Consumer。

➢ **消费者集群(Consumer Cluster)**:集群中的每个 Consumer 都与 Broker(Topic 的提供者)建立长连接,且需要支持向 Broker 的主从节点订阅消息。
  - 集群中每个 Consumer 订阅的 Tag 必须一样。
  - Consumer 每隔一段时间向 Broker 发送 Pull 消息的请求,Broker 收到这些请求后如果有消息数据则立即向 Consumer 返回,否则就会阻塞直到有消息数据才会返回。
  - Consumer 接收到消息数据后再调用上层消费者进程设置的 listen 方法。

➢ **缓冲区集群(Broker Cluster)**:Broker 提供了 Topic 和队列机制来进行一定程度的持久化工作,同时支持 Push 和 Pull 两种消息推送模式,也支持主从结构的容错机制。每个 Broker 与 NameServer Cluster 中的一个节点维持长连接关系,定时将 Topic 的信

息注册到所有 NameServer 中(通知 Topic 的路由)。

➢ **名称服务集群(NameServer Cluster)**:NameServer 提供了轻量级的服务发现和路由机制,每个 NameServer 都记录了完整的 Topic 路由信息,并支持快速存储扩展。

从这些组件的描述中可以总结出,Producer 是消息的生产商,Consumer 是消息的最终消费者,而 Broker 则是一个尽职尽责、不赚差价的中间商,这三者中传递的消息数据无论走哪条路都要征求 NameServer 的意见。

在海量数据场景下,Consumer 会通过 RebalanceService 线程每隔一段时间做一次基于 Topic 的队列负载均衡,对于这些队列中的消息,可以按照平均分配的策略进行均衡化拉取。

本章小结

通信在 Windows 系统中占有非常重要的地位。

Windows 原生就支持了若干通信模型,这些模型的核心任务就是事件的多路分离。在这个大前提下,根据工作机制的不同又可以分为 I/O 完成端口模型、select 模型和重叠 I/O 模型三大类。这些模型是应用软件中各种用户态通信框架的基础。

应用软件一般使用通信框架负责通信和传输工作,这是因为通信框架具有良好的封装性和易用性。在 Windows 通信框架中,ACE 算是最庞大、最复杂、最通用也是最完善的通信框架了。除此之外还包括专门用于互联网媒体通信的 WebRTC 框架、用于 Web 前端的 Netty 框架以及分布式微服务环境下的 Istio 框架等。通信框架针对具体的业务场景侧重点各有不同。

消息队列作为一种能够“削峰填谷”的通信组件,其承担的任务越来越重要,特别是在海量数据高并发的场景下,消息队列是必备的。目前开源和商用的消息队列非常多,资料也很全面,本文没有事无巨细地都列举出来。

第22章　新一代通信技术

新一代通信技术方兴未艾,特别是在5G即将到来的大环境下,为了应对高带宽海量数据传输,需要有新的通信技术作为支撑,以打破传统通信设备和技术的瓶颈与壁垒。

新的通信技术从软件方面来说包括两个方向,一个是软件定义网络(SDN),另一个则是高性能通信协议栈,后者(例如DPDK、SPDK等)是前者的基础支撑。

在本章中我们按照图22－1所示的提纲从两个方向予以展开,首先介绍SDN的相关技

- 通信技术
  - 软件定义网络
    - SDN
      - 问题的提出
        - 传统网络在5G时代有什么壁垒和问题？
        - SDN的定义和技术特点，在5G时代能解决什么问题？
      - SDN的框架
        - SDN数据平面的设备、工作机制
        - SDN控制平面的接口、工作机制
        - SDN应用平面的位置
        - （SDN的核心）
      - SDN的内部机制
        - OpenFlow协议
          - OpenFlow演进历史
          - OpenFlow报文种类
        - 流表
          - 什么是流表？流表在不同版本的OpenFlow协议中有什么异同？
          - 流表的工作机制与处理过程
        - 组表与计量表
      - 链路发现：LLDP的工作机制
      - SDN的开源和闭源项目（控制器），OpenDaylight等
      - SDN的衍生
        - SD-WAN
        - IBN
    - NFV（以此为基础）
      - 什么是NFV？NFV的重要名词解释
      - ETSI定义的NFV参考架构
      - ETSI定义的NFV接口体系
    - 网络切片（以此为基础）
      - 什么是网络切片？为什么要网络切片？
      - 网络切片的分类和部署方式
      - 网络切片的生命周期管理
  - 数据平面加速技术
    - 等同于高性能通信协议栈
    - DPDK框架
      - 现有网络协议栈的瓶颈与壁垒、DPDK与传统协议栈驱动的异同
      - DPDK框架的关键技术
      - DPDK框架的结构和工作流程
      - DPDK框架报文处理的两种模型：流水线模型和RTC模型
    - SPDK框架
      - 基于DPDK的成熟框架和模块，用于NVMe SSD存储驱动栈
      - 预备知识
        - 什么是RDMA？RDMA的优势是什么？
        - RDMA服务的架构和基于RDMA的三种传输方式
          - IB、iWARP、RoCE
        - InfiniBand协议及其机制
        - RoCE协议及其机制
        - iWARP协议及其机制
        - （支持的OSI参考层是不一样的）
        - 什么是HDD和SSD，各有什么特点？
        - 什么是NVMe？NVMe的工作机制是什么？与存储主机等要素之间的关系是什么？
      - SPDK的设计特点
        - 1. 线程与处理器核的管理角度
        - 2. 线程间通信的角度
        - 3. I/O处理无锁化机制的角度
      - SPDK的框架特点：自底向上包括了三层，这三层的作用是什么？

图22－1　本章提纲

术和衍生产品,再介绍基于数据平面加速技术的高性能通信协议栈。

22.1 软件定义网络

随着5G通信、大数据和云计算等技术的发展,传统网络设备中数据转发与策略控制集成在一起的传输机制越来越不适应数据量剧增带来的变化。特别是随着5G技术及其应用的全面展开,传统交换设备已不能适应数据转发、路由寻址、QoS保障等业务,并造成了以下问题:

- **路由转发协议过于复杂**:在传统网络中,由于数据转发面(数据平面)与路由控制面(控制平面)被集成到一台设备中,作为传输设备要处理IGP、BGP、MPLS等多种封包和路由协议,不能全心全意为数据转发服务。
- **路由和转发策略调整难度很大,不能适应业务层面频繁变化的需要**:传统传输设备的路由和转发策略是配置在本地设备中的,对于QoS等需要经常改变路由和转发策略的业务及服务的响应不灵活。
- **数据平面与控制平面耦合度高**:传统传输设备将数据平面与控制平面集成在一起,使设备成为一个个"堡垒",用户只能成套地采用固定厂家的设备,绑架了用户与厂家,不利于网络生态的建设。

上述问题在5G时代来临之际会更加尖锐。在这种情况下,SDN和NFV诞生了。2006年斯坦福大学首先提出了OpenFlow的概念,通过OpenFlow实现网络的可编程能力。在这种思想的指引下,SDN(Software Defined Network,软件定义网络)应运而生。正如其字面意思,在OpenFlow的催化作用下,SDN实现了网络能力的可编程化、网络基础设施的通用化、数据平面与控制平面的解耦化和路由策略下发的灵活化。其中网络能力的可编程化是SDN的基础,其发展历史如图22-2所示。

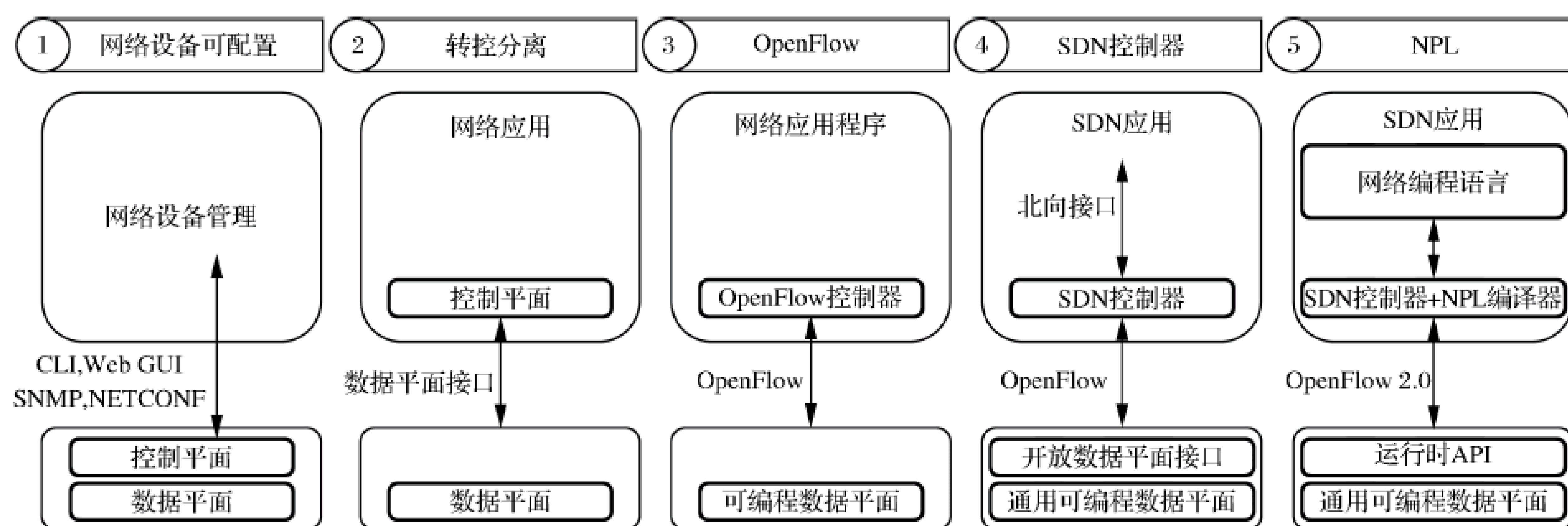
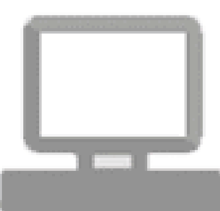


图22-2 网络可编程发展历史

NFV则在SDN的基础上更进一步提出了网络功能虚拟化的概念,瞄准的是网络实体的虚拟化使用。NFV(Network Function Virtualization,网络功能虚拟化)于2012年由美国AT&T、英国电信等网络运营商在ETSI(欧洲通信标准协会)提出。NFV旨在基于X86/X64等架构的通用硬件基础上使网络功能抽象化、服务化、软件化、与硬件解耦化,并基于虚拟化





技术使网络服务运行于虚拟机之上,进一步提升了通用硬件的利用率和网络效能。

22.1.1　SDN

SDN 的主旨是通过软件来定义网络,很巧妙地解耦了传输设备的数据平面和控制平面,因此带来了如下的好处:

- 数据平面可以专心致志地负责转发数据的工作,其处理网络包的栈深大大降低。例如原先可能要解析到第三层甚至第四层协议,并通过查表来决定如何处理网络包,解耦之后数据平面可能只需要解析到第二层并通过预先由控制平面(SDN 控制器)下发的转发策略就可以知道往哪里转、转给谁。
- 控制平面(SDN 控制器)需要处理的数据也大大减少了,数据平面只有在遇到陌生的、先前没有遇到过的网络包才会打扰咨询控制平面如何处理网络包。控制平面会通过 OpenFlow 协议将此类包的转发策略下发给数据平面,自己则悠闲地"继续度假"。
- 控制平面对外发布北向接口(例如 Restful),供上层网络管控和业务系统使用。也可以发布东、西向接口并与其他异构网络进行对接,以支持其他异构网络的互联互通。这样做大大简化了上层业务系统的开发难度,使之不再需要与具体的传输协议打交道了。

SDN 体系主要分为两个技术派别。

- **开放网络基金会(Open Networking Foundation,ONF)**:包括德国电信(DT)、Facebook、Google、微软、Verizon、Yahoo、日本 NTT 电信、高盛等。
- **Linux 基金会(Linux Foundation)**:包括思科(Cisco)、IBM、微软、Big Switch、博科、思杰、戴尔、爱立信、富士通、英特尔、瞻博网络、微软、NEC、惠普、红帽和 Vmware 等,并提出了开源项目 OpenDaylight。

22.1.1.1　SDN 框架结构

SDN 最大的特点就是数据平面和控制平面的分离(转发与控制分离,简称转控分离),传输交换设备将其中的控制面板(决定了如何路由转发)"提升"到了 SDN 控制器,两个层面采用 OpenFlow 协议进行通信,如图 22-3 所示。

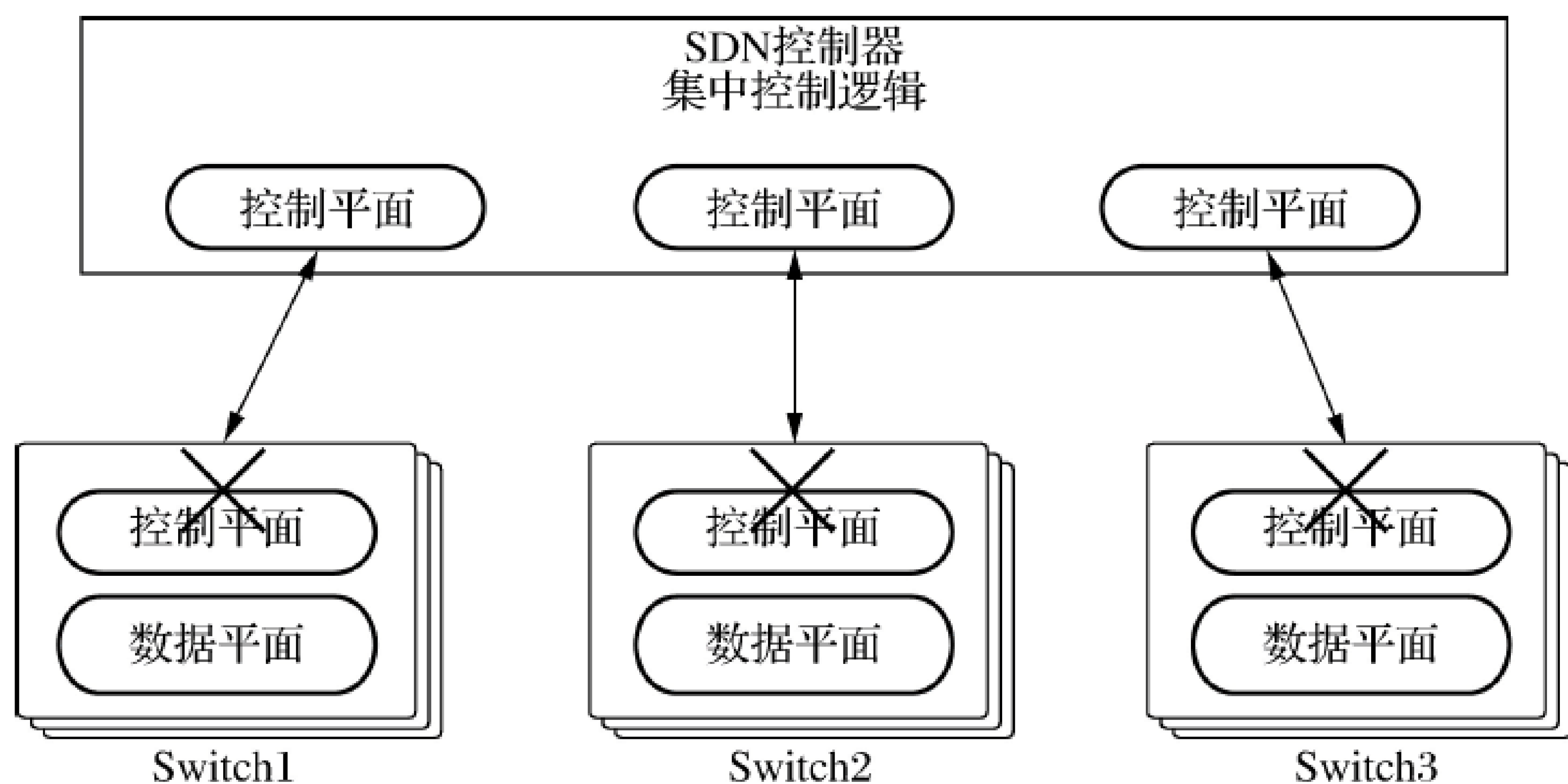


图 22-3　转控分离的 SDN 框架结构

1. SDN 数据平面

传统交换机要做的事情就是在 OSI 参考模型的二层或三层转发网络包,包括转发策略的学习/更新以及以太帧转发等功能。在 SDN 中,转发策略的学习和更新这种更偏向于控制的行为是由 SDN 控制器完成的,SDN 数据平面交换机只是负责转发表(流表)的维护和以太帧转发的工作。在 SDN 体系下,数据平面就是指 SDN 交换机。SDN 交换机可以分为纯 SDN 交换机、混合交换机、白盒 SDN 交换机等类型。

➢ **纯 SDN 交换机**:只支持 OpenFlow 协议,负责数据包转发和内部流表维护,所有流表均由 SDN 控制器下发。
  - 数据包进入交换机时,交换机查找流表以确认是否有流表项匹配。
  - 若有流表项匹配成功,则执行该表项指定的操作;若无,则查看是否已设置丢弃。
  - 若已设置丢弃,则丢弃此数据包;若没有设置丢弃,则根据设置转发该数据包至 SDN 控制器,待控制器下发对应流表项后根据此流表项进行相关操作。

➢ **混合交换机**:支持 OpenFlow 协议和传统的转发协议,适配性更强。

➢ **白盒 SDN 交换机**:所谓白盒 SDN 交换机就是交换机硬件(芯片 + 机体 + 操作系统)和交换机软件可分离的 SDN 交换机,用户可以自行选择软硬件,以获得最大的收益。但白盒 SDN 交换机要求操作系统(例如开源的 OpenSwitch)具有良好的兼容性,既能向下兼容芯片和其他硬件,也能向上承载所有软件应用。可以看出,白盒 SDN 交换机是软硬件分离在交换机中的具体实践。白盒 SDN 交换机概念的产生得益于 2011 年由 Facebook 牵头成立的 OCP(Open Compute Project)项目,该项目着眼于开发数据中心基础设施架构。OCP 的 Open Network 子项目则定义了交换机的硬件和芯片的 SAI 及版本管理标准,使得交换机硬件具有了可替换性。

前文提过,SDN 交换机的控制一般采用 OpenFlow 协议。虽然 OpenFlow 协议只是 SDN 交换机中应用最广泛的控制协议,不是唯一的控制协议,但在 SDN 的协议日益规范化的大趋势下,OpenFlow 已经默认成为 SDN 交换机的标准控制协议。因此在本文中,我们也称 SDN 交换机为 **OpenFlow 交换机**,称 SDN 控制器为 **OpenFlow 控制器**,就是为了突出它们的协议属性。从是否仅支持 OpenFlow 协议的角度看,OpenFlow 交换机可分为以下两类:

➢ **OpenFlow-Only Switch**:仅支持 OpenFlow 转发。

➢ **OpenFlow-Hybrid Switch**:既支持 OpenFlow 转发,也支持普通的二、三层转发。

OpenFlow 交换机采用了虚拟化的设计思想,一个交换机可以支持多个 OpenFlow 实例,每个实例相当于一个虚拟机环境,因此每个实例也可以被看作一台独立的虚拟交换机。每个 OpenFlow 实例可以单独连接 OpenFlow 控制器,因此一个 OpenFlow 交换机也可以同时被多个 OpenFlow 控制器控制。

除了控制协议,SDN 交换机的配置一般采用 OF-Config 协议。OF-Config 协议于 2012 年由 ONF 组织发布,其主要目标是在支持 OpenFlow 的网络设备上实现基本功能配置,例如配置一个或多个控制器的 IP 地址、设备队列及端口等资源,支持远程修改设备的端口状态等。

SDN 交换机在收到网络包分组时首先对包头进行解析，匹配交换机内部的流表，并根据流表规定的动作处理所收到的网络包，其处理流程如图 22－4 所示。

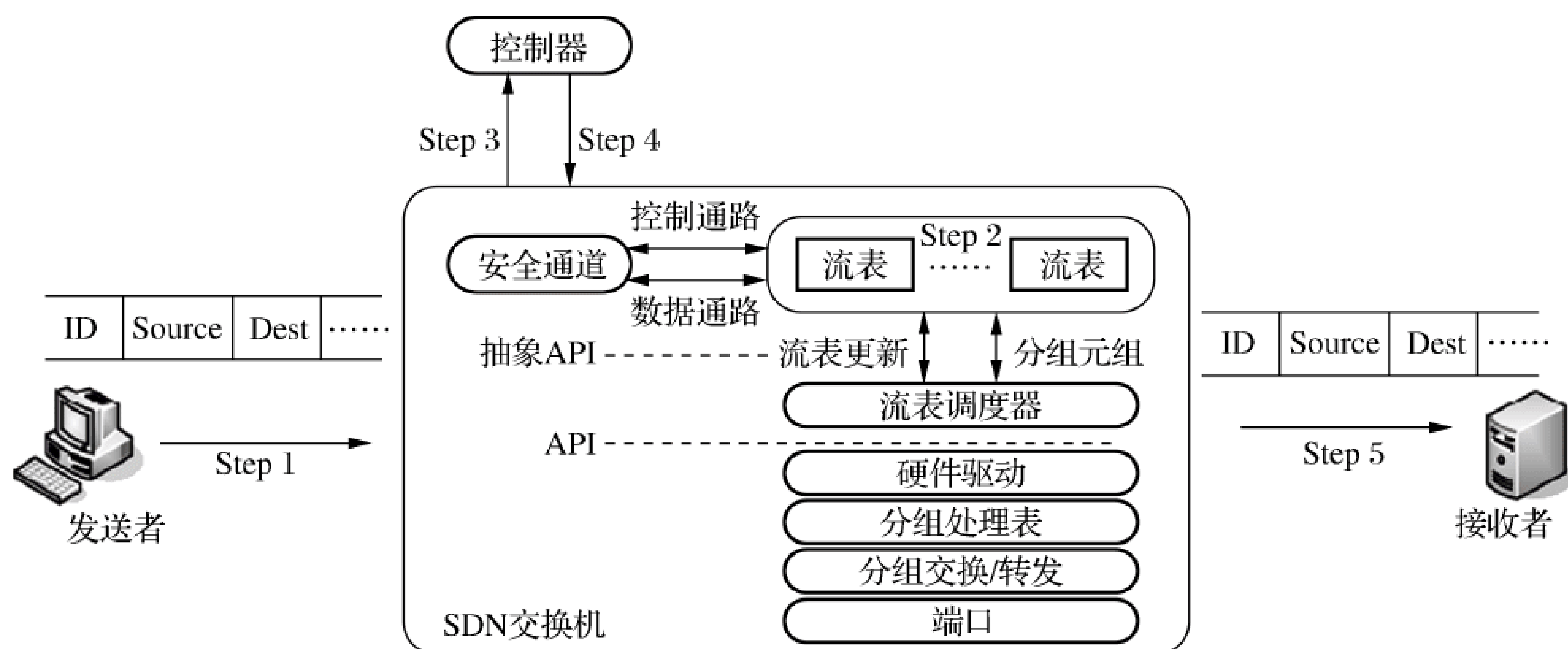


图 22－4　SDN 交换机的工作流程

（1）SDN 交换机接收到网络包分组时解析分组头，并在流表中查询该分组是否有流和动作的对应规则。

- 如果有对应规则，则转发到相应的端口或丢弃该分组（丢弃也是一种动作），流程结束。
- 如果没有对应规则，SDN 交换机将该数据分组封装为 OpenFlow 的 Packet-in 报文并上传至 SDN 控制器。

（2）SDN 控制器收到 Packet-in 报文后向 SDN 交换机下发对应的转发策略流表项。

（3）SDN 交换机将该流表项加入流表中，并将步骤（1）中接收到的网络包及后续同类型网络包（比如相同五元组或相同的 IN 端口）转发到指定的端口中。

在解析分组头时，如果为二层转发则只需要解析以太帧的帧头；如果为三层转发则需要解析到 IP 包的包头；如果为更高层的传输应用则可以继续向上解析四层、五层的包头。这些解析动作的规则是由 SDN 控制器制定和下发的，而真正的决策者是比 SDN 控制器更上层的网络传输业务，SDN 控制器只是一个“中层领导”，负责翻译和下发决策者的指令。

在 SDN 交换机与控制器交互的过程中，为了保证数据传输的安全性和不可破译性，可以采用安全传输层协议（TLS）对 OpenFlow 协议进行加密封装，构筑安全通道。如果安全通道采用 TLS 连接加密，当交换机启动时，会尝试连接到控制器的 6633 TCP 端口（OpenFlow 端口默认的建议设置为 6633），双方通过交换证书进行认证。因此，在加密时，每个交换机至少需配置两个证书。

交换机可以通过简易的队列机制为 QoS 提供有限的支持。每个交换机端口可以关联若干个队列，这些队列可以提供队列缓冲等机制或功能，并可以通过外部配置工具对这些队列进行配置。在交换机中，每个队列都由一个对应的数据结构描述，该数据结构包含了队列 ID、队列关联的端口、最小传输速率保证、最大传输速率限制、计数器等成员变量。OpenFlow 的流表动作中有个 Set-Queue 动作，就是用于将一个队列流表项映射到已配置好的端口上，

当网络报文分组经过该流表项时就被转发到对应的队列中了。

OpenFlow 1.3 更是定义了计量表(Meter Table),用于触发各种与性能相关的动作并作用在流上。

2. SDN 控制平面

SDN 控制器是控制平面的具体执行者,也是 SDN 网络架构的核心。控制器是一个全局的集中式网络控制单元,具有高屋建瓴般的视野和格局。一般的路由选择控制都是局部的分布式,路由器在自治系统之中或之间通过内部网关协议(Interior Gateway Protocol,IGP)或外部网关协议(Exterior Gateway Protocol,EGP)发现路由拓扑,并基于此计算最短路径。但无论是 IGP 还是 EGP,在遇到网络阻塞和中断的时候,都是由离故障节点最近的路由器最先感知,且无法立即向全网进行通告。这也是非集中式控制系统的通病。一般情况下,SDN 控制器在一个 SDN(SDN 域)中只有一个(灾备的不在此考虑之列),当遇到阻塞和中断时,下层的 SDN 交换机会将信息通告给 SDN 控制器,在遇到要流经故障节点的流量时,控制器会通过更改流表项动作的方式将其集中调度到别的转发路径上。

SDN 控制平面的任务包括:

- ➢ 接收上层应用程序的网络控制请求并翻译成 SDN 数据平面的特定指令。
- ➢ 通过 OpenFlow 等协议向 SDN 交换机下发流表项以控制数据流走向。
- ➢ 通过东西向接口联通同构或异构的外域网络。
- ➢ 路由选择,负责收集路由和 QoS 信息(例如节点连通性和传输的网络时延等)并计算最短路径。
- ➢ 收集 SDN 交换机的告警信息,处理 SDN 交换机上报的陌生报文分组。
- ➢ 集中管理 SDN 交换机资源,并维护交换机之间的连接拓扑信息,支持通过 OF-Config 等协议配置交换机。
- ➢ 集中管理 SDN 交换机的流量信息,收集统计流量数据。
- ➢ 向应用层提供功能支持。包括提供拓扑结构、网络状态、网络统计等统计分析类功能,也包括网络配置管理、网络监控、网络故障排除、网络安全策略设置等网络自动化运维相关的应用。

SDN 控制器包含东、西、南、北 4 个方向的接口,其中东、西向接口为非必要接口,只在存在多个网络的情况下才需要;南、北向接口为必要接口,南向的协议面向 SDN 交换机,一般为 OpenFlow(用于控制)和 OF-Config(用于配置),北向接口供上层应用和云管理平台控制 SDN 网络,一般为 Restful API,如图 22-5 所示。

SDN 诞生的初衷之一就是实现数据平面与控制平面解耦,打破目前控制器与数据转发器被少数厂商垄断的窘境。不过从目前的情况来看,SDN 控制器厂商大多仍是采用私有协议与 SDN 交换机通信互联,并没有实现彻底的解耦。这也是通信行业的现实情况。

除了南、北向协议,SDN 的东、西向协议针对的是网络域之间的互联互通。不过东、西向接口也没有固定的协议或者标准可以遵循。每个 SDN 域就是一个自治系统,每个自治系统

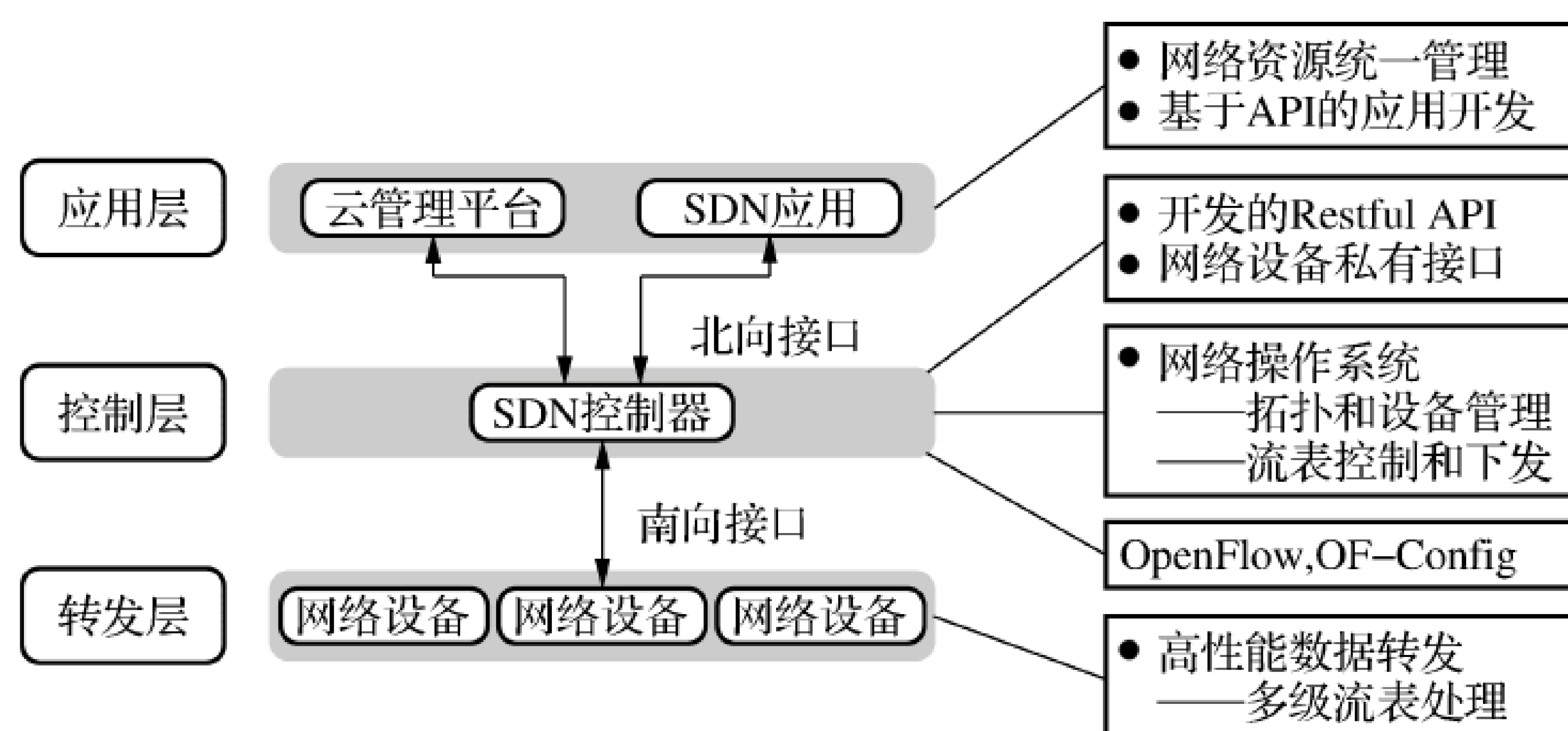


图 22－5　SDN 控制器的南、北向接口

具有一个 SDN 控制器，当然 SDN 控制器下的域可大可小。东、西向协议不仅要处理同构域（都是 SDN 域）的互联，也要处理异构域（不都是 SDN 域）的互联。因此，SDN 控制器必须具有 BGP（边界网关协议，是基于 TCP/IP 的互联网的默认外部路由协议）的能力并且能够获取到相邻 BGP 实体的位置信息，其工作过程如图 22－6 所示，大致如下：

（1）SDN 控制器启动并触发和激活 BGP 模块。

（2）BGP 模块向每个相邻的 BGP 实体（可能是其他域中 SDN 控制器的 BGP 模块，或是非 SDN 域中的网关设备）创建 TCP 连接，相邻的 BGP 实体被称为"邻居"。

（3）TCP 连接创建完成后，BGP 模块向邻居发送 Open 报文，通过 Open 报文交换路由能力信息，交换完成后一条 BGP 连接也就建立成功了。

（4）BGP 模块向已经建立 BGP 连接的邻居发送 Update 报文，以交换彼此的网络层可达性信息。选择 SDN 控制器之间的最合适路径时使用该可达性信息和 QoS 信息等，并且将可达性信息更新到控制器本地的路由选择信息库（RIB），以便对数据平面的 SDN 交换机设置流信息。

（5）通过 RIB 中的多条可用路径进行路由选择，创建最佳可用路由。当有网络包分组需要在 SDN 域间穿越时可以向 SDN 交换机下发由该路由生成的流表项。



图 22－6　SDN 控制器的东、西向路由选择过程

3. SDN 应用平面

除了 SDN 数据平面与控制平面,还有作为最上层的 SDN 应用平面,它也是 SDN 框架的重要组成部分。SDN 应用平面基于 SDN 控制器提供的 Restful API 等接口,实现了资源管理、路由器业务、访问控制、负载均衡、应用加速、QoS 业务、流量工程、安全防御、虚拟资源管理和网络运维与配置等功能,并向运营商用户展示网络拓扑结构图,如图 22－7 所示。

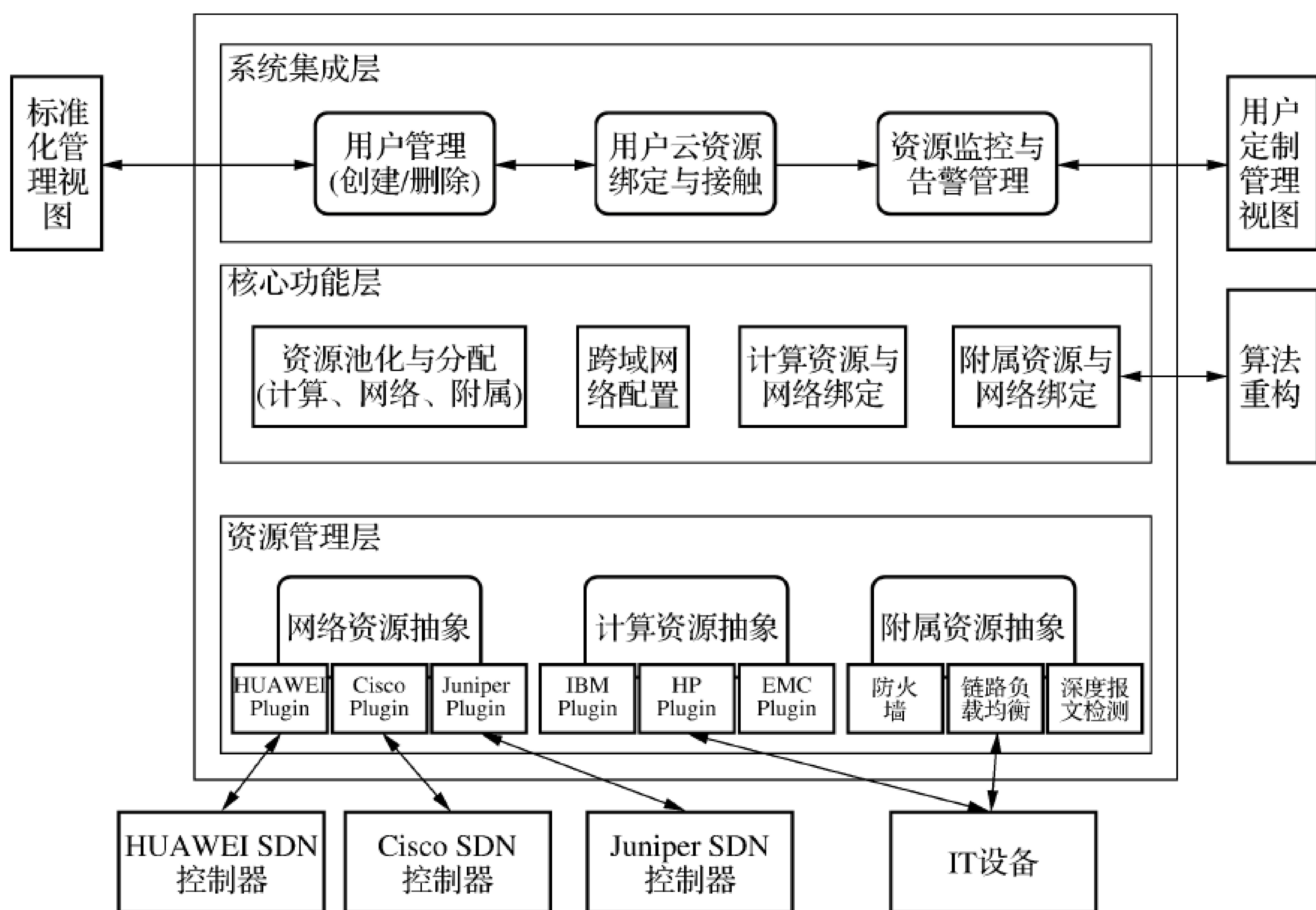


图 22－7　SDN 应用平面架构视图

22.1.1.2　OpenFlow 协议与相关表结构

OpenFlow 协议(OpenFlow 信道)、流表和组表是 OpenFlow 交换机最重要的三大组件,构成了交换机的核心,如图 22－8 所示。下面我们将对这些组件分别进行介绍。

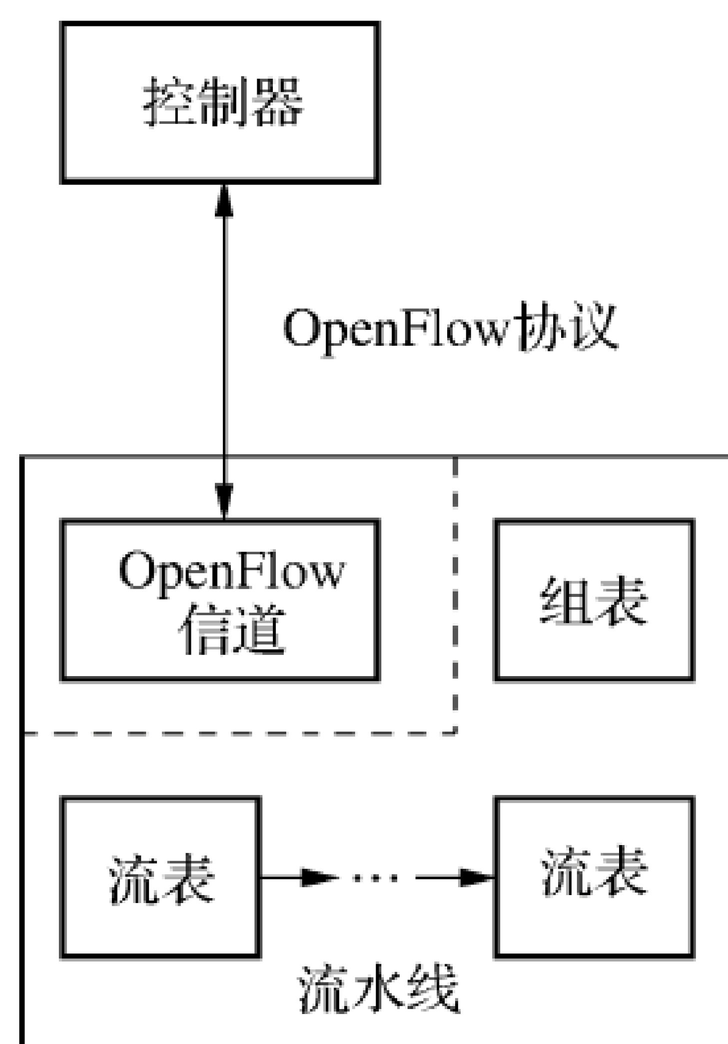


图 22－8　由流表、组表、OpenFlow 协议组成的 OpenFlow 交换机

1. OpenFlow 协议

OpenFlow(OF)协议始于 2009 年,由斯坦福大学首先提出,针对的就是网络转控分离的架构解耦。OpenFlow 经历了若干个版本,这些版本都是由 ONF(开放网络基金会)组织来制定的,每个版本都是对上一个版本的修正与补充。作为 SDN 控制器南向接口的主要实现方式,可以说 OpenFlow 的历史(如图 22－9 所示)也代表了 SDN 的发展史。

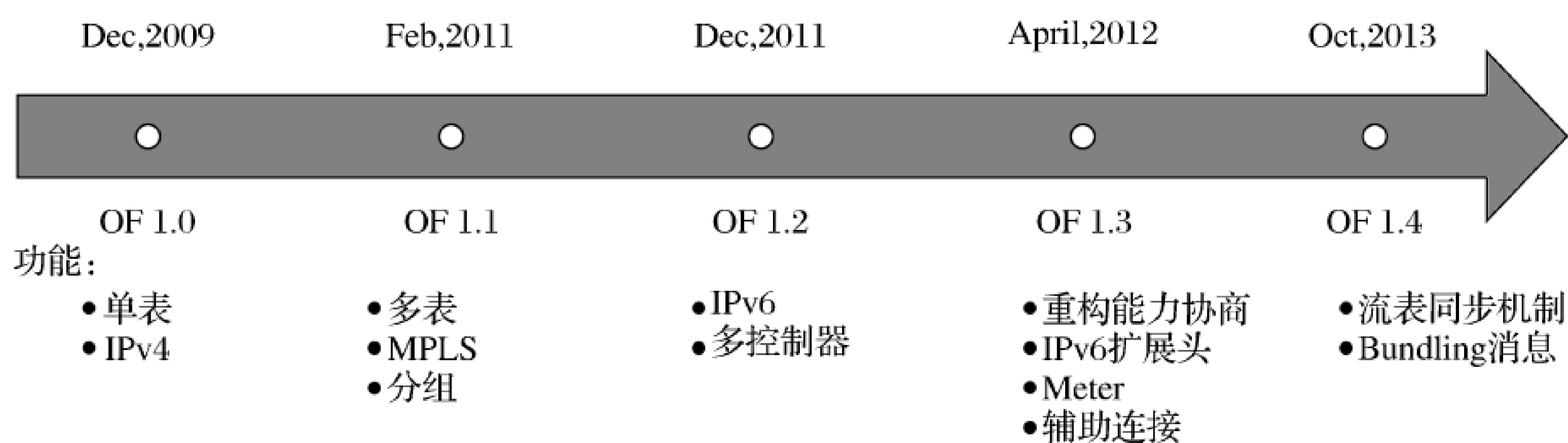


图 22－9　OpenFlow 协议演进路线图

OpenFlow 是 SDN 控制器与 SDN 交换机之间的接口，这个接口也被称为 OpenFlow 信道。OpenFlow 信道可以采用 TCP 协议以明文方式传输，也可以采用 TLS 协议以密文的方式传输。但为了传输的安全可靠，OpenFlow 信道一般采用后者。

OpenFlow 端口用于 SDN 交换机之间建立逻辑连接，网络报文分组进入和离开 OpenFlow 流水线都要经过 OpenFlow 端口。端口包括三种类型，即物理端口、逻辑端口和保留端口。

- **物理端口**：交换机与以太网对应的端口。
- **逻辑端口**：不直接与物理端口对应，而是一种高层的抽象端口，是需要高层应用和协议来定义的。一个逻辑端口可以对应多个物理端口。逻辑端口一般与业务类型相关的网络报文分组相对应，例如链路聚合、隧道、环回地址等，可能多个物理端口都是服务于隧道业务的，那就将这些物理端口都划分为一个逻辑端口。
- **保留端口**：这是由 OpenFlow 协议规定的通用性的转发动作，包括 ALL、CONTROLLER、TABLE、IN_PORT、ANY、LOCAL、NORMAL、FLOOD 等动作。

OpenFlow 协议是 SDN 控制器与 SDN 交换机之间的控制协议，允许对交换机流表中的流表项进行增、删、改的操作，表 22－1 列出了具体的控制协议报文，可分为如下三类：

- **控制器到交换机的报文**：这类报文均由控制器产生，在部分情况下不要求交换机的响应，其作用是管理交换机的逻辑状态，例如流表项和组表项的配置等。
- **异步报文**：这类报文由交换机产生，用于向控制器报送自身状态和未知网络报文分组。
- **对称报文**：除上述两类报文外，对称报文是能起到辅助作用的报文，例如控制器与交换机之间的握手报文、带宽度量报文等。

表 22 - 1　OpenFlow 报文

报文类型	报文名称	报文描述
控制器到交换机的报文	Features	控制器向交换机请求功能信息,交换机需将自身的功能信息返回给控制器
	Configuration	设置与查询配置参数,交换器需要发送响应报文
	Modify-State	对流表项和组表项进行增、删、改,同时也包括设置交换机的端口属性
	Read-State	收集交换机的信息,包括当前配置信息、统计信息、功能信息等
	Packet-out	将报文分组引导至交换机特定端口上
	Barrier	屏蔽请求/响应报文,用于控制器保证消息的依赖性和接收消息的完整性
	Role-Request	设置和查询 OpenFlow 信道的角色,多用于交换机被连接到多个控制器的场景
	Asynchronous-Configuration	对异步报文设置过滤器或查询过滤器信息,多用于交换机被连接到多个控制器的场景
异步报文	Packet-in	将网络报文分组发送给控制器
	Flow-Removed	将流表中流表项的删除信息通报给控制器
	Port-Status	向控制器通报交换机端口变化信息
	Role-Status	当交换机从主控制器变为从控制器时,将这种角色的转变通报给控制器
	Controller-Status	OpenFlow 信道状态改变时通知控制器,当控制器失去通信能力时会协助进行故障恢复
	Flow-Monitor	将流表的变化通报控制器,以便控制器实时监控其他控制器对流表的修改
对称报文	Hello	控制器与交换机之间的握手协议报文
	Echo	控制器与交换机都可以发出 Echo 请求报文,对方需要回复 Echo 响应报文
	Error	控制器或交换机将自身的问题和故障通报给对方
	Experimenter	用于在 OpenFlow 未来的版本中添加新的功能

从表 22 - 1 中可知,交换机可以为控制器提供事件类报文、流统计信息类报文和采用 Packet-in 报文封装后的陌生报文分组,控制器可以通过这些报文管理 SDN。

SDN 控制器与 SDN 交换机建立连接的过程是这样的:

(1) 交换机向控制器建立 TCP 连接。

(2) 连接建立后,交换机与控制器相互发送 Hello 报文,用于在两者之间协商版本。

(3) 控制器向交换机发送 Feature Request 报文,请求获取交换机性能、功能等参数。

（4）交换机向控制器返回 Feature Reply 报文，报文中附带着交换机的详细信息。

（5）交换机与控制器交换完上述信息后，彼此定时发送 Echo Request 和 Echo Reply 这两个心跳保活报文。

交换机与控制器的连接断开后，交换机会进入 Fail Open 模式，该模式有两种类型：

- **Fail Secure Mode**：在该模式下，OpenFlow 交换机的流表项继续有效，直到超时后才会被删除，即支持流表项的正常老化。
- **Fail Standalone Mode**：所有报文分组均通过保留端口正常处理，OpenFlow 交换机退化为普通以太网交换机。不过这种模式只适用于 OpenFlow-Hybrid 类型的交换机，因为除 OpenFlow-Hybrid 类型之外的交换机不支持以太网功能。

2．流表（Flow Table）

OpenFlow 交换机内部有许多被称为流表的数据结构，用于对经过交换机的网络报文分组进行处理。每个分组在交换机内都会经过一个或者多个流表，每个流表包含多行，每一行被称为一个流表项（Flow Entry），流表项中规定了每一条网络报文分组的匹配特征、流向等信息。流表的字段结构是与 OpenFlow 的版本号相关联的，越往后的版本中流表结构越复杂，如图 22－10 和 22－11 所示。

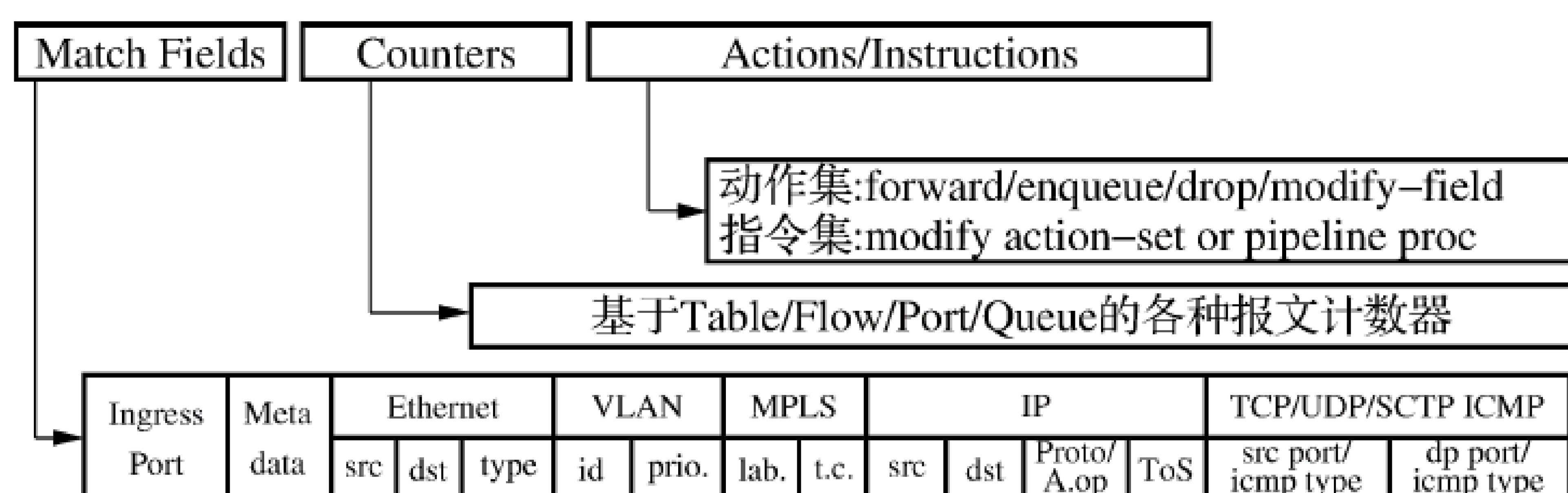


图 22－10　OpenFlow 1.0 流表项与其匹配字段的结构

Match Field	Priority	Counter	Instruction	Timeout	Cookie

图 22－11　OpenFlow 1.3 流表项的结构

从图中可以看出，OpenFlow 1.3 的流表项比 OpneFlow 1.0 的流表项在结构上增加了若干属性：

- **Priority（优先级）**：用于标志流表匹配的优先顺序，优先级越高匹配越早。默认优先级为 0，针对的是报文分组匹配多条流表项的情况。同一个流表中按流表项的优先级进行排序匹配。
- **Timeout（超时时间）**：用于表示该流表项老化的时间，超时了就删除，以节省内存资源。
- **Cookie（附属信息）**：表示由控制器选择的不透明数据值。控制器用来过滤流统计数据，包括流改变和流删除的数据，但处理数据包时不能使用。

流表项中的匹配字段（Match Fields）是一组网络报文协议头域的组合，用于标识该表项

对应的流；计数器(Counter)对匹配的报文分组进行记录，包括每个端口的接收分组数、每个端口或队列的传输分组数、持续时间等统计信息；当匹配上了某条表项后就可以执行表项后的动作/指令(Action/Instruction)，这些指令如表 22－2 所示。

表 22－2　OpenFlow 1.3 流表项指令集

指令类型	指令说明	可选/必选
Write-Actions	更改动作集(Action Set)中的所有动作	必选
Goto-Table	进入下一级流表	必选
Meter	对匹配到流表项的网络报文分组进行限速	可选
Apply-Actions	立即执行动作	可选
Clear-Actions	清除动作集中的所有动作	可选
Write-Metadata	更改流表间数据，在支持多级流表时使用	可选

要注意的是指令集中的指令必须遵照一定的顺序执行，即：

- Meter(meter_id)指令必须在 Apply-Actions 指令之前执行；
- Clear-Actions 指令必须在 Write-Actions 指令之前执行；
- GoTo-Table 指令必须在最后执行。

因此指令执行的优先顺序为：Meter→Apply-Actions→Clear Actions→Write-Actions→Write-Metadata→Goto-Table。每个流表项的指令集中每种指令类型最多只能有一个。

动作集(如表 22－3 所示)是与通过流表流水线的网络报文分组相关联的，当分组通过各个流表时，流表会指示对其进行各种处理进而形成动作集，这些动作在分组即将离开流水线时会被统一执行。流表项中的处理动作也允许对报文分组进行修改，当匹配下一个流表时会按照修改后的新报文分组进行匹配。

表 22－3　OpenFlow 1.3 流表项动作集

动作类型	动作说明	可选/必选
Output	转发到指定的 OpenFlow 端口	必选
Drop	无直接动作，指令集中无 Output 动作则丢弃该报文	必选
Group	处理报文到指定分组	必选
Set-Queue	设置报文的队列 ID	可选
Push-Tag/Pop-Tag	写入/弹出标签，例如 VLAN、MPLS 等	可选
Set-Field	修改报文的头字段	可选
Change-TTL	修改 TTL、Hop Limit 等字段	可选

OpenFlow 交换机中允许存在多个流表(OpenFlow 1.3 版本后支持的功能)，这些流表串起来就形成了流表流水线(pipeline)。所有流表依次排列，从序号 0 开始，网络报文分组依次通过这些流表。每个流表都包含若干流表项，流表项中的指令如表 22－2 所示。显然，前者是

将报文分组转到下一个流表中处理,后者是对报文分组直接处理,包括 Output、Drop、Set-Queue 等。流表的流水线处理过程如图 22 - 12 所示。

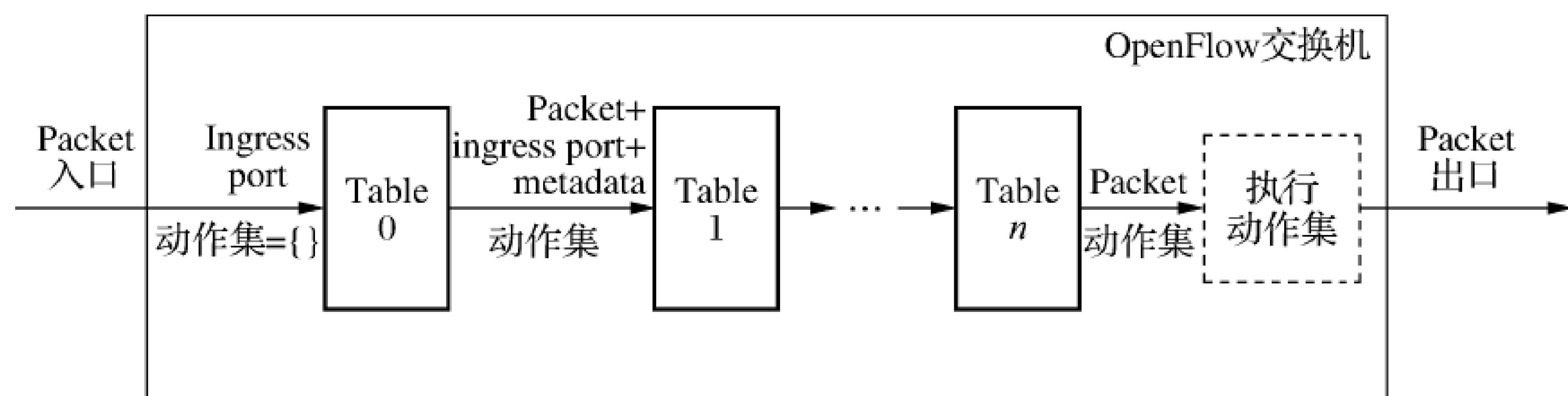


图 22 - 12　流表的流水线处理过程

流表中通常默认会有 table miss(表未命中)这个流表项。table miss 表项的优先级最低,为 0,且不匹配所有流表项。该流表项的作用是:在某个流表中找不到和该报文分组匹配的流表项时就交由 table miss 表项来处理,通常的处理动作是转发给 SDN 控制器、丢弃掉或转交给后面的流表。

当网络报文分组进入交换机时,首先从 0 号流表开始匹配。流表的处理顺序必须按照序号从小到大进行处理,不能"逆流而动"。当分组匹配某一流表项时,首先会更新该流表项的统计计数,再根据流表项中的指令进行相应操作,例如转到下一个流表或立即执行动作集等。当报文分组流动到最后一个流表时,其对应的动作集会被全部执行。

流表是由 OpenFlow 控制器下发的,有两种下发模式:

- **主动模式**:控制器将自己的流表信息主动下发给交换机。这种方式比较占用交换机的内存空间,但处理效率也比较高。不过为了节省空间,一般情况下控制器只是发送初始的基本流表而非全部流表。
- **被动模式**:交换机收到陌生网络报文分组(没有匹配的流表项)时,将该报文分组上交给控制器,控制器向该交换机下发这条分组的流表项。被动模式下交换机无需维护大量流表,只在遇到陌生报文时按需索取。

3. 组表(Group Table)

组表本质上也是一种流表,由若干个组表项(Group Entry)组成。组表通过指向不同的组来为报文提供转发目标。这些分组的类型包括:All、select、Indirect 和 Fast Failover,其中 All 和 Indirect 是交换机必须支持的两种类型。图 22 - 13 是组表项的结构。

Group Identifier	Group Type	Gounters	Action Bucket

图 22 - 13　OpenFlow 组表项的结构

- **Group Identifier(组 ID)**:表示分组的唯一标识,是个 32 位整数。
- **Counter(计数器)**:表示使用该分组的次数。
- **Action Bucket(动作桶)**:表示包含多个动作的列表。
- **Group Type(分组类型)**:即列出的 4 种类型,每种类型的解释如下:
  - **All**:依次执行动作桶中所有的动作,多用于组播和广播,分组中的每个动作桶都会

复制一份网络报文分组，每个桶都关联一个不同的输出端口。

- **select**：会根据交换机事先被设置的算法（例如散列算法等）来执行某个桶中的动作，例如可以按照控制器设置的权重算法来进行发送的负载均衡。
- **Indirect**：执行一个已在分组中定义的动作桶。这种分组中只有一个动作桶，多用于多条流分组的聚合输出，例如只改变桶中的输出目的地址就可以改变与该桶关联的多条流的输出方向。
- **Fast Failover**：这种类型的分组可以让交换机在无需向控制器发起查询的情况下就修改转发路径，多用于端口故障快速恢复的场景。

引入组表的目的是为了支持 L2/L3 多路径转发，同时也支持链路聚合、ECMP、快速重路由、BGP 下一跳汇聚等扩展的额外能力。

4. 计量表（Meter Table）

计量表提供了报文分组的限速功能，流表项通过引用计量表的表项实现了该功能，每个表项的结构如图 22－14 所示。

Meter Identifier	Meter Band	Counter

图 22－14　OpenFlow 计量表项的结构

一个计量表表项可以包含一个或者多个 Meter Bands，每个 Meter Band 定义了速率以及动作。若报文的传输速率超过了某些 Meter Band，则根据这些 Meter Band 中速率最大的那个定义的动作进行处理。

22.1.1.3　SDN 链路发现机制

在传统网络中，链路发现一般都是由转发单元（网元）自主进行的，但在 SDN 中，链路发现则是由具有全局视野的 SDN 控制器统一完成的。

链路层发现协议（Link Layer Discovery Protocol，LLDP）是一种标准的链路发现方式，可以将本地端交换设备的主要信息（地址信息、以太网类型、设备标识、接口标识等）以 LLDPDU（链路层发现协议数据单元）的形式向周围的直连单元发布和扩散。对于 OpenFlow 交换机来说，这些单元自然也包括与之相连的 OpenFlow 控制器，也就是说 OpenFlow 控制器采用 LLDP 进行链路发现。LLDP 链路发现的一般过程如下：

（1）控制器向所有与之相连的交换机发送一个 Packet-out 报文，该报文中封装了 LLDP 数据包，以命令交换机将 LLDP 数据包发送给所有端口。

（2）交换机收到 Packet-out 报文后将该报文中的 LLDP 包发送给与该交换机相连的所有交换机（包括控制器），也就是邻居交换机。

（3）如果邻居交换机是 OpenFlow 交换机，就会针对该报文查询自己的流表并进行处理。由于邻居交换机没有专门的流表项处理该 LLDP 数据包，因此只能将其封装成 Packet-in 报文并返回给控制器。

（4）控制器收到该 Packet-in 报文后进行分析，并在其保存的链路表中添加上述两台交

换机的连接记录,如此便生成了一条链路。其他交换机也照此办理,最终生成全局的连接链路。

（5）如果步骤(3)中的邻居交换机不是 OpenFlow 交换机,也就是说 OpenFlow 交换机连接着非 OpenFlow 域,此时控制器会要求所辖交换机发送广播包,除控制器外的所有端口所连交换机都能收到该广播包。

（6）广播包在非 OpenFlow 域中穿过,最终到达与该非 OpenFlow 网络域相邻的另一个 OpenFlow 域的交换机(OpenFlow 交换机)中。该交换机必然没有对应的流表项处理该广播包,自然也会将其封装为 Packet-in 报文并上交给自己所属的控制器,从而形成另一条新的链路。

22.1.1.4　SDN 控制器项目

1. 现有 SDN 控制器的开源和商业项目

SDN 控制器既有商业项目,也有开源项目,比较知名的控制器项目包括:

- **OpenDaylight(ODL)**:由 Cisco 和 IBM 创立的开源 SDN 控制器项目,主要成员都是网络设备厂商,并由 Linux 基金会主持。
  - 这是一种用于网络可编程的 Java 语言的平台,实现了单一集中式控制器,并具有很强的扩展性和适配性,同时也实现了与 NFV 开发平台 OPNFV、OpenStack 和开放网络自动化平台 ONAP 的同步。
  - OpenDaylight 控制器是以化学元素来命名的,最初的产品是 Hydrogen(氢),最新的版本是第 8 个版本:Oxygen(氧)。
- **开放网络操作系统(ONOS)**:全称是 Open Network Operating System,是由 AT&T 等运营商和其他服务提供商支持的首款开源的 SDN 网络操作系统。
  - ONOS 于 2014 年发布,并得到了开放网络基金会的支持。
  - 该项目主要用于分布式控制器,可靠性强、性能好、灵活度高,主要面向运营商和企业主干网。
- **NOX/POX**:作为第一款真正意义上的 SDN OpenFlow 控制器,NOX 支持 OpenFlow 1.0 和 C++ 的 API,并且采用了异步的基于时间的编程模型。
  - POX 是 NOX 的更新版本,支持 Windows、Linux 等系统。
  - POX 是一种由 SDN 从业者支持的开源 OpenFlow 控制器,使用 Python 语言开发,具有良好的 API 文档和友好的 Web 用户界面。
- **Floodlight**:由 Big Switch Networks 公司研发的开源项目,支持 Restful API。
- **Ryu**:由 NTT 实验室研发的开源 SDN 框架,基于 Python 语言开发。
- **Onix**:由 VMWare、Google 和 NTT 共同研发的分布式 SDN 控制器商业项目。
- **其他商业控制器**:包括 Cisco 的 CiscoOne、华为的 SNC(Smart Network Controller,智能网络控制器)、Brocade 的 Vyatta SDN 控制器、Juniper 的 OpenContrail、华三的 VCF (Virtual Converged Framework,虚拟融合架构)控制器等。

虽然大多数开源控制器是基于 OpenFlow 协议开发的,但大多数商业控制器却是基于开源代码和各厂商的私有协议开发的,因此实现交换机与控制器的真正解耦还很难。

2. OpenDaylight

OpenDaylight 是一套模块化、可插拔、易于扩展的开源 SDN 控制器平台架构,该架构采用 Java 语言开发,并基于 Java 的动态模型系统 OSGi(Open Service Gateway initiative,开放服务网关倡议)框架实现了一个组件化的、基于模型编程的控制器,如图 22-15 所示。

OpenDaylight 控制器主要包括开放的北向 API、控制器平面、南向接口和协议插件。北向 API 分为 OSGi 和 REST 两类,其中 OSGi 是有状态连接,有注册机制,而 REST 是无状态连接。同一系统进程内的应用使用 OSGi 类 API,而不同系统进程之间的应用则使用 REST 类 API。上层应用程序可以利用这些北向 API 获得网络拓扑信息和运行算法等并进行分析,支持设计部署新的网络策略。

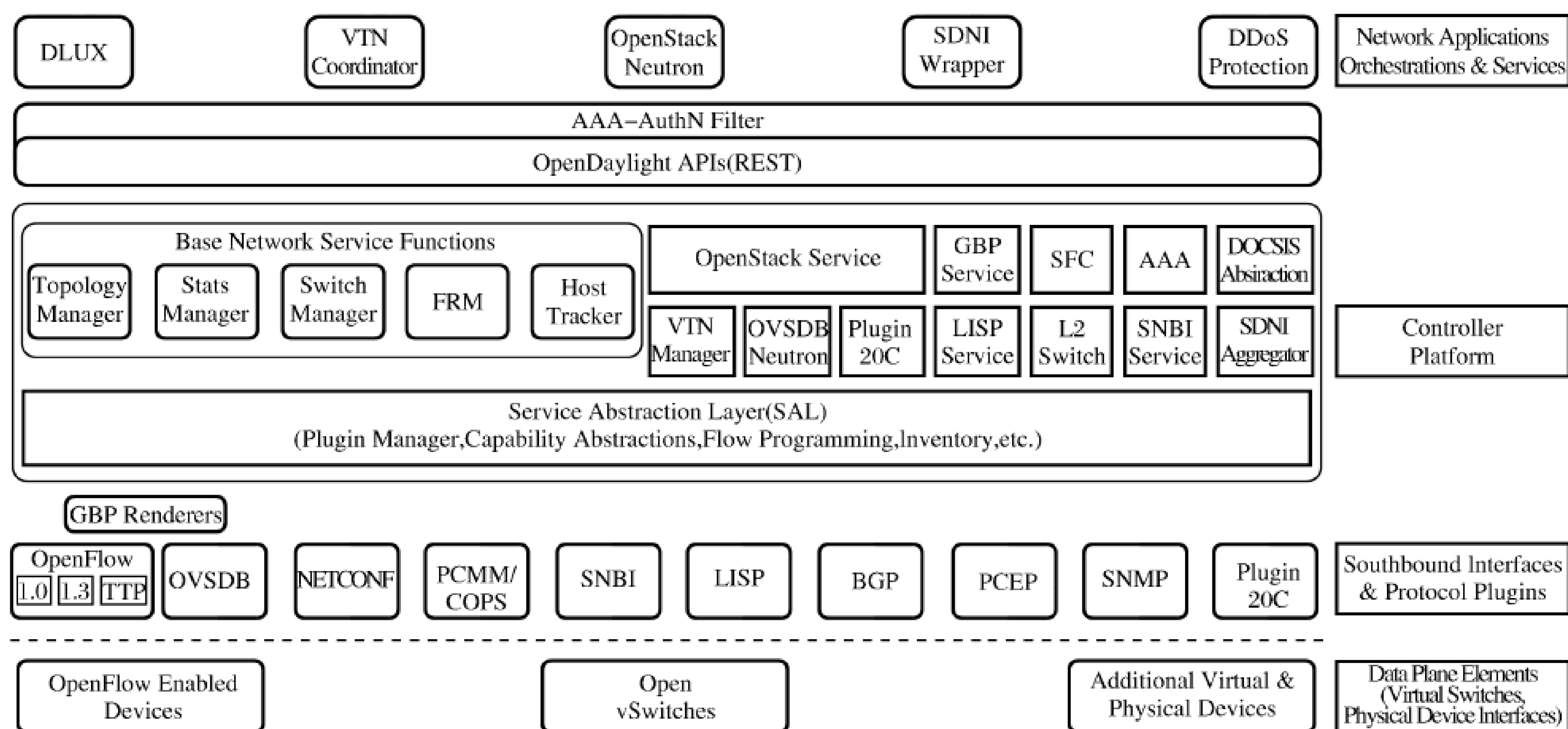


图 22-15　OpenDaylight 控制器平台架构(图片来自 SDNLAB)

控制器平台包括一系列功能模块,可动态组合以提供不同的服务,这些功能模块包括拓扑管理、转发管理、主机监测、交换机管理等模块。

服务抽象层(Service Abstraction Layer,SAL)是控制器模块化的核心,用于自动适配底层不同的设备,使开发者专注于业务应用的开发。SAL 北向连接功能模块,以插件的形式为之提供底层设备服务;南向连接多种协议插件,屏蔽不同协议的差异性,为北向功能模块提供一致性服务,因此 SAL 起到中间调度的作用。

南向接口支持多种不同协议,如 OpenFlow 1.0、OpenFlow 1.3、Netconf、BGP-LS(拓扑发现和收集)、PCEP(路径计算单元通信协议)等,用于将控制策略下发到设备,并从底层的转发设备获取数据。这些底层设备支持混合模式交换机和经典 OpenFlow 交换机。

目前的 OpenDaylight 控制器平台引入了 MD-SAL(Model Driven-SAL,模型驱动 SAL),用于替换早期的 AD-SAL(API Driven-SAL,API 驱动 SAL)。两者在设计上有以下不同:

➤ MD-SAL 引入了 Data Store,用于存储上层应用和下层设备所定义的 Yang 模型数据。

Yang 是随 Netconf 协议产生的数据建模语言，类似于 XML Schema 和 SNMP SMI，具有良好的可读性和可扩展性。

- MD-SAL 中北向和南向插件均可以对上述 Yang 模型进行存取。
- MD-SAL 中北向和南向插件的适配是由单独的适配插件完成的，当北向和南向的 API 是一一对应的关系时，可以采用访问共同的 Data Store 的方式实现数据交互而无需像在 AD-SAL 中那样注册和绑定 API。
- 在 MD-SAL 中提出了数据提供者和消费者的概念，二者均需要向 MD-SAL 注册。
- 在 MD-SAL 中，数据消费者向 MD-SAL 订阅数据，在收到数据提供者的 Notification 消息后可以从 Data Store 中获取数据。
- AD-SAL 非常类似 Windows 中的 NDIS 框架。在 NDIS 中，上层与下层的驱动均向 NDIS 适配层注册 API，以使上层协议栈驱动可以调用下层 NIC 驱动的接口，同样下层 NIC 驱动也可以回调上层的协议栈驱动的接口。在 AD-SAL 中亦是如此，南向和北向插件可以分别调用对方注册的 API，由 Request Routing 完成二者的消息传递。
- 在 MD-SAL 中具有南向和北向两个 Yang 模型，分别面向上层服务和下层设备。南向和北向插件都通过模型自动生成的 API 来存取 Data Store 内的数据。
- 在 MD-SAL 中提供了三种访问 Data Store 的方式：Binding API、Binding Independent API 和 HTTP RestConf API。

AD-SAL 和 MD-SAL 的架构对比如图 22 - 16 所示。

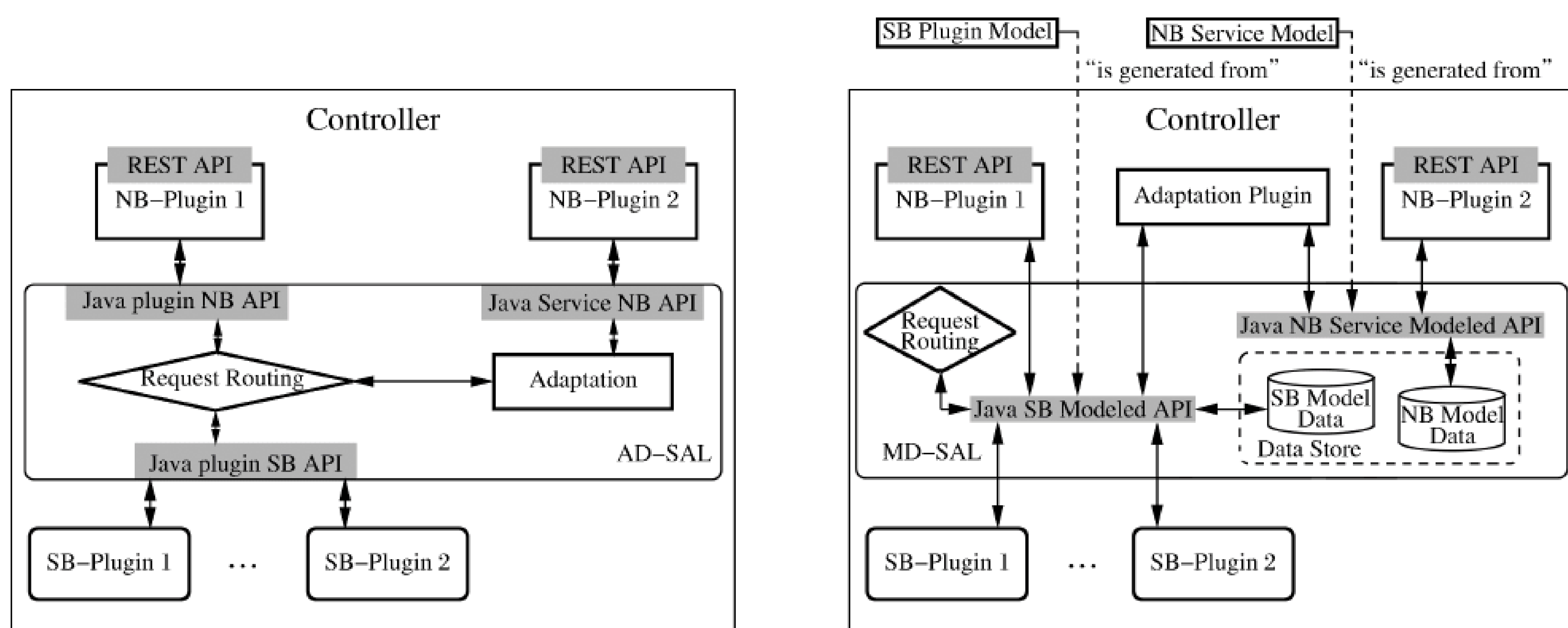


图 22 - 16　AD-SAL 与 MD-SAL 的架构对比(图片来自 CSDN)

MD-SAL 本身不提供任何模型，所有的模型都是由对应的插件来提供的，MD-SAL 只是提供了相应的注册机制以及插件间的连接机制。

SAL 起到了承上启下的中间调度作用，也就是完成了请求的路由过程。其北向连接功能模块，以插件的形式为之提供底层设备服务；南向连接多种协议，屏蔽不同协议的差异性，为上层功能模块提供一致性服务，使得上层模块与下层模块之间的调用相互隔离。SAL 可自动适配底层不同的设备，使开发者专注于业务应用的开发。

因此,MD-SAL 层本质上就是一个消息传递组件,负责为上下两层的各个插件提供消息转换服务。在 OpenDaylight 中消费者通过 RPC 方式调用提供者的服务,提供者可以向消费者发送 Notification 消息(消费者需要事先订阅),使得 Data Store 的读写和数据变化可以被通知到各个组件。这些工作都是由作为中介的 MD-SAL 完成的。

综上所述,OpenDaylight 控制器整体上采用了如下的设计原则:

- **运行时模块化和扩展化(Runtime Modularity and Extensibility)**:支持在控制器运行时进行服务的安装、删除和更新。
- **多协议的南向支持(Multiprotocol Southbound)**:南向支持多种协议。
- **服务抽象层**:南向多种协议对上提供统一的北向服务接口。Hydrogen 中全线采用 AD-SAL,Helium 版本中 AD-SAL 和 MD-SAL 共存,Lithium 和 Beryllium 版本中已全部使用 MD-SAL 架构。
- **开放的可扩展北向 API**:通过 REST 或者函数调用方式提供可扩展的应用 API,两者提供的功能是一致的。
- **支持多租户和切片(Support for Multi-tenancy/Slicing)**:允许网络在逻辑上或物理上被划分成不同的切片或租户。控制器的部分功能和模块可以管理指定切片,控制器根据所管理的切片来呈现不同的控制观测面。
- **一致性聚合(Consistent Clustering)**:提供细粒度复制的聚合和确保网络一致性的横向扩展(scale-out)。

22.1.1.5 SD-WAN

WAN 就是广域网。一个区域内的局域网(LAN)一般都会通过路由器等设备接入 WAN 中。在不同的 WAN 之间的连接包括有线和无线两种,有线的服务如 MPLS、T1、承载着以太网或者商业宽带的链路等;无线的服务则包括蜂窝数据网络(如 4G LTE)以及公共 Wi-Fi 或者卫星通信。在 WAN 领域多采用 MPLS(多协议标签转换)的方式来实现 VPN。

SD-WAN(Software-Defined WAN,软件定义广域网)将特定硬件组成的 WAN 网络的控制能力通过软件方式“虚拟化”,利用集中控制器实现设备的抽象与统一调度,是对 SDN“转控分离”思想的继承。也就是说 SD-WAN 与 SDDC(软件定义数据中心)一样,是 SDN 的一种应用,其根本目的就是利用多条现网链路,根据业务应用的优先程度综合选择线路质量,自动选择最佳路径,保证流量成本、负载均衡和网络质量(取代 MPLS-VPN、IPSEC-VPN 等)。特别是基于应用的路由选择和优化、DDoS 安全防护等是现阶段 SD-WAN 最为广泛的应用实例。虽然 SD-WAN 和 SDN 的思想同出一脉,但由于应用环境不同,实现方式也是迥异的。与 MPLS 相比,SD-WAN 不依赖于专用的传输设备,也不需要事先做专门的配置,因此具有更好的灵活性和开通的便捷性。

根据 SD-WAN 控制器对网络域的控制边界,可以将 SD-WAN 划分为 4 种架构:

- **叠加架构**:实现了多接入控制方式,并且 SD-WAN 控制器控制了边缘设备到网关设备的上行流量,也简化了 WAN 的操作管理。

➢ **云端架构**：实现了多个接入点（PoP）的集成，在每个接入点都部署了虚拟边缘路由器（vPE），路由器一侧与各分支的边缘设备建立 VPN 隧道，另一侧与 ISP 提供的 MPLS 设备直连。汇聚上来的流量可通过多个隧道转发到 MPLS 主干网中。
➢ **整合架构**：网络管理部件可以管理多个运营商的边缘路由器（PE）而不仅仅是多个边缘设备，提高了混合组网的能力。
➢ **原生架构**：SD-WAN 控制器统一管理边缘设备、网关设备、骨干核心网 PE 等，全网统一编排、统一调度。

按照功能分界和转控分离的原则，可以将 SD-WAN 自底向上地划分为三个主要的层次：

➢ **SD-WAN 边缘设备层**：对应于 SDN 数据转发层，负责 SD-WAN 隧道的连接和拆除，包括隧道绑定、网络优化、数据包传输控制、应用程序识别、数据缓存等基础层的功能，支持常见的 SDN 南向协议。
➢ **SD-WAN 控制器层**：对应于 SDN 控制平面，通过与边缘设备层建立安全连接来执行策略的下发，包括路由协议的下发和边缘设备状态的获取。因此控制器层的功能模块包括应用传输路径的动态调整、实时监测和告警、流量采集与分析等。
➢ **SD-WAN 服务编排层**：对应于 NFV 中的 VNFM，负责整个服务生命周期的管理与编排，因此包括策略定义模块、审计和实施模块、安全模块、模板管理模块等，并生成了一个个诸如流量控制链、传输加速链这样的服务链（Service Chain）下发给控制器层，控制器层会根据这些服务链生成对应的流表并继续向转发层下发。服务编排层同时也提供了北向接口，供业务和计费层的软件调用。

22.1.1.6　IBN

IBN（Intent-Based Networking，基于意图的网络）的核心理念是可以对用户侧的业务策略进行翻译并转换为必要的网络配置，继而自动实施、运维和纠错。因此，IBN 可被看作帮助网络管理员规划设计、实施部署和操作运维以提高网络灵活性和可用性的软件系统。Gartner 提出的 IBN 的定义有以下 4 部分内涵：

➢ **转译和验证**：系统从最终用户处获取更高级别的业务策略，并将其转换为必要的网络配置，生成并验证最终的设计和配置以保证正确性。
➢ **自动化实施**：系统可以在现有网络基础设施上配置适当的网络变更，配置可以通过网络自动化或网络编排完成。
➢ **网络状态感知**：系统为其管理控制下的业务系统提供实时网络状态，这些操作都是协议和传输不可知的。
➢ **保障和自动化优化补救**：系统持续验证原始业务意图是否正确实现，并且可以在意图无法实现时采取纠正措施。

因此，IBN 本质上就是能让网络管理员得到他们想要的网络的一种工具，并且兼具了网络运维和自动纠错的能力。网络管理是个复杂的活儿，目前的网络管理软件是通过命令行界面进行配置的，这就要求精确的输入和小心翼翼地维护，并且需要随时应对由于操作不慎

而造成的断网情况。IBN 可以将网络管理员从这些烦琐的工作中解脱出来,并且会持续验证原始的业务需求是否被满足,当不被满足时可以随时纠偏,这样就会形成一个持续闭环的系统,提升了可用性和敏捷性。

基于上述定义,IBN 的运行过程大致是这样的:收集意图→转译和验证→自动下发与执行→自我优化补救→实时反馈,这也是 IBN 运行的状态机。

- **收集意图(Intent Collection)**:意图就是用户希望达成的网络状态,这个状态在转译生效之前可以灵活修改。
- **转译和验证(Translation and Validation)**:将网络意图转化为网络策略,网络策略就是网络能够理解的语言了。生成策略后要根据网络状态实时验证有效性,如果验证不通过则再从收集意图开始执行。
- **自动下发与执行(Automated Implementation)**:通过验证的策略会由 IBN 系统下发到基础网络上自动执行。
- **自我优化补救(Self-Remediation)**:由于网络的状态是动态变化的,因此执行时与验证时的状态可能不一致,IBN 系统会自动根据先前的意图对策略进行优化和纠偏。
- **实时反馈(Post-Assessment)**:将运行结果反馈给 IBN 的发起者(网络管理员等),由发起者研判是否已经符合最初的意图。

总之,IBN 是一种基于人类意图去部署和实现的网络,并支持自我纠偏机制。这种网络最直接、最理想的实现策略就是软件定义,因此 IBN 也是 SDN 的延伸和发展,也许未来 IBN 会成为网络发展的大趋势。

22.1.2 NFV

NFV(Network Function Virtualization,网络功能虚拟化)是由 ETSI 的 NFV 工作组提出的一种技术,也是一种部署和管理网络服务的全新方式。其核心思想是通过 X86/X64 等通用计算平台和 Linux 等通用操作系统,结合虚拟化和云化技术来承载网络功能软件,一方面使网络功能软硬件解耦分离,另一方面也可以提高硬件设备中网络软件的运行效率,同时还可以实现网络业务的快速迭代和自动部署,实现弹性伸缩、故障隔离和自动修复等功能。可以说 NFV 是希望借助于云计算、容器化和微服务的思想,实现网络业务的微服务化,实现自动运维,并最终实现一套开放的网络平台。正因为采用了虚拟化技术,NFV 的名称也由此而来。不过,由于运行于虚拟机上,性能必然会有损耗。特别是目前 NFV 的相关生态和软件都不是十分成熟,NFV 的商用化道路还很长。

首先来看一些相关名词的解释。

- **VNF(Virtualized Network Function)**:虚拟网络功能,主要是指路由器、防火墙、深度包检测(DPI)、NAT 等网络设备功能的软件化。包括 VNF(虚拟网络功能)和 EMS(单元管理系统,用于对 VNF 的功能进行配置和管理)两部分,一般情况下 EMS 和 VNF 是一一对应的。

- **NFVI(Network Function Virtualized Infrastructure)**:网络功能虚拟化基础设施,提供 VNF 的运行环境,包括所需的硬件及软件。其中硬件包括计算、网络和存储资源;软件包括 Hypervisor、网络控制器、存储管理器等工具。NFVI 将物理资源虚拟化为虚拟资源供 VNF 使用。
- **PNF(Physical Network Function)**:物理网络功能,是指在专用硬件上的控制平面与数据平面集成在一起的传统网络设备。
- **CNF(Cloud-Native Network Function)**:云原生网络功能,是指容器化的 VNF 和可以微服务化的容器网络和服务网格。
- **VIM(Virtualized Infrastructure Manager)**:虚拟化基础设施管理器,NFVI 管理模块通常运行于对应的基础设施站点中,主要功能包括:资源发现、虚拟资源管理分配、故障处理等,为 VNF 的运行提供资源支持。
- **VNFM(VNF Manager)**:VNF 管理器,主要对 VNF 的生命周期(实例化、配置、关闭等)进行控制,一般情况下与 VNF 一一对应。
- **NFVO(NFV Orchestrator)**:NFV 编排器,是 NS(网络服务)生命周期的管理模块,同时负责协调 NS、组成 NS 的 VNF 以及承载各 VNF 的虚拟资源的控制和管理。
- **OSS/BSS**:OSS 是指通用管理系统(Operation Support System),BSS 是指业务支撑系统(Business Support System),两者都是服务提供商的管理功能,不属于 NFV 框架内的功能组件,但 NFVO 需要提供对 OSS/BSS 的接口。
- **网络切片(Network Slicing)**:在单一的物理基础设施之上运行多个虚拟网络。
- **MANO(Management And Orchestration)**:管理与编排,NFV-MANO 由 NFVO(NFV 编排器)、VNFM(VNF 管理器)、VIM(虚拟化基础设施管理器)等组件构成。

22.1.2.1　NFV 参考架构

我们再来看一下 NFV 的参考架构。NFV 参考架构已经由 ETSI 给出,如图 23－17 所示,包括以下几个部分:

- **通用管理和业务支撑系统**:包括 OSS 和 BSS,承担了基础设施和虚拟网络功能的管理,并与 NFV-MANO 交互网络服务中的各种信息,是整个 NFV 架构中最接近用户的一层。
- **网络服务系统**:包括了网元管理(EM)和虚拟网络功能(VNF)两大部分,其中 EM 负责 VNF 的创建、配置、监控等管理工作。
- **网络功能虚拟化基础设施(NFVI)**:包括 NFV 系统中的物理和虚拟资源,其中物理资源包括计算、网络、存储;虚拟资源是对物理资源的抽象,用以承载 VNF。
- **网络功能虚拟化管理与编排(NFV-MANO)**:包括了 NFV 编排器、VNF 管理器和虚拟化基础设施管理器三大部分。

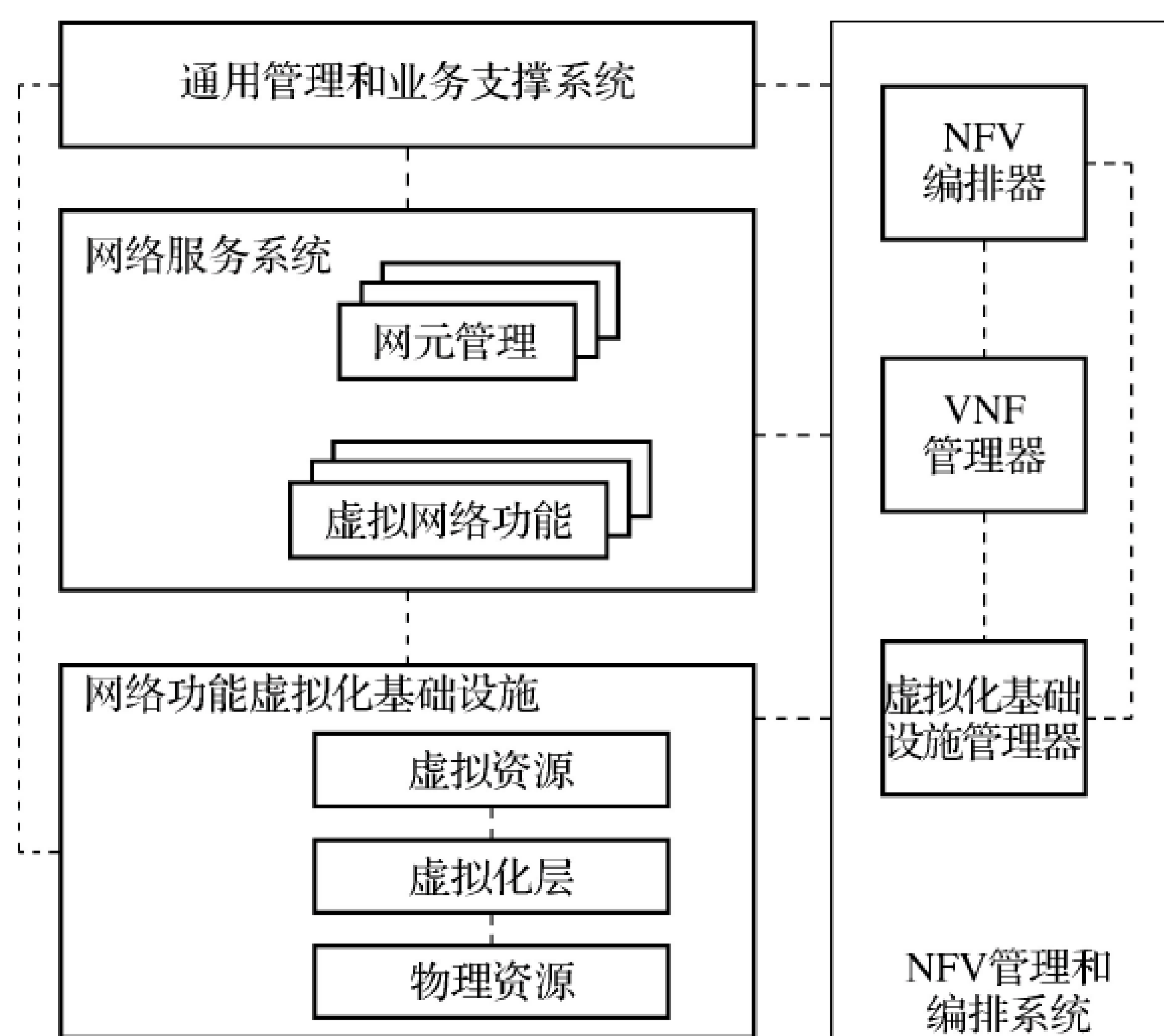


图 22－17　ETSI 定义的 NFV 参考架构

更进一步地来看,NFV-MANO 是整个 NFV 架构中的核心,包括了 NFV 编排器(NFVO)、VNF 管理器(VNFM)和虚拟基础设施管理器(VIM)三个基本组件,各个组件具有以下特性。

1) NFVO

➢ NFVO 是 NFV-MANO 中最重要的组件,是 NFV 的编排引擎。所谓编排就是对虚拟化资源和基于虚拟化资源之上的软件的管理。
➢ NFVO 可以进一步分为服务编排器(Service Orchestrator)和资源编排器(Resource Orchestrator)两个部分。服务编排面向网络功能服务,资源编排包括软件资源管理和虚拟机资源管理。
➢ NFVO 包括一个北向接口 OS-NFVO 以及两个南向接口 NFVO-VNFM(表示 NFVO 与 VNFM 之间的接口)和 NFVO-VIM。
➢ OS-NFVO 是面向运营商 OSS/BSS 和 MANO 之间的接口,包括网络服务的管理接口,例如描述符、生命周期、性能、告警和 VNF 包等。
➢ NFVO 也支持验证并授权 NFVI 的资源请求。

2) VNFM

包括了 VNF 生命周期管理、性能管理、告警管理、指标管理和操作权限管理等功能的接口,即对 VNF 进行创建、删除、扩展、故障诊断和恢复等操作,并上报运维数据。

3) VIM

VIM 管理虚拟化层(例如 OpenStack 等),可以将 VNF 部署到不同的云上,支撑 VNF 间的协作。常见的资源虚拟化平台包括 OpenStack、Vsphere、AWS、Azure 等。

目前也有许多 MANO 的开源框架,包括但不限于下列项目:

➢ **ONAP 项目**:开放网络自动化平台,是由 AT&T 和中国移动主导的开源项目,成员包括了诸多运营商和设备厂商。

- **OSM(Open Source MANO)项目**:ETSI 领导的并由运营商驱动的 MANO 开源项目。
- **OPNFV 项目**:运营商级的开源参考平台,包括构建 NFVI、VIM 等,并将 API 包含在了其他 NFV 元素中。

22.1.2.2　NFV 接口体系

ETSI 也为 NFV 各层之间定义了接口体系,如图 22－18 所示。

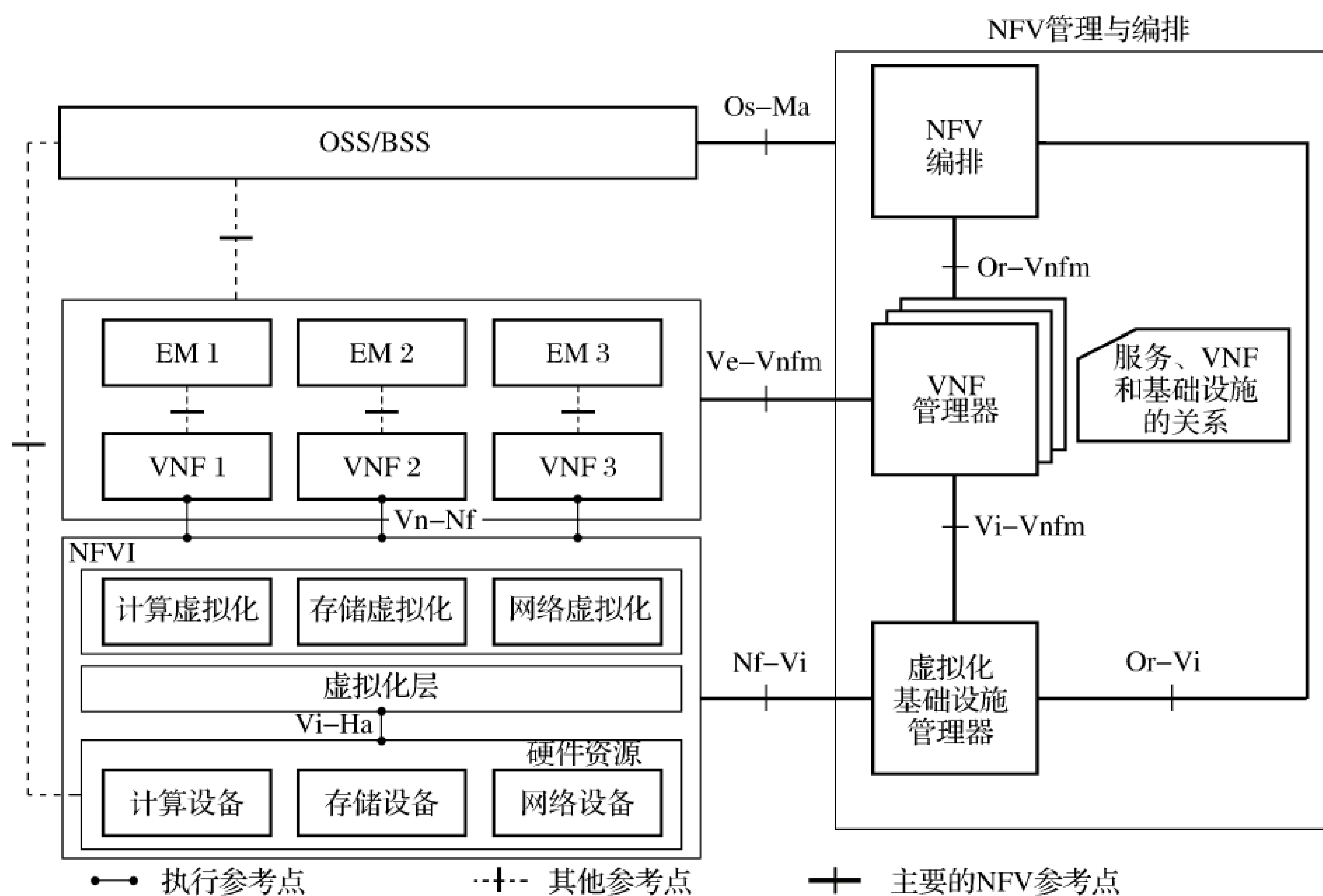


图 22－18　ETSI 定义的 NFV 接口体系

- **Vi-Ha(Virtualization Layer↔Hardware Resources)**:提供了虚拟化层与硬件资源层之间的接口,可以按照 VNF 的要求分配计算、存储和网络资源实体,同时收集硬件设备的信息上报到虚拟化层以便于对硬件平台进行运维和监控。
- **Vn-Nf(VNF↔NFV Infrastructure)**:底层硬件被虚拟化以后与 VNF 之间的接口。这是个抽象接口,并没有实质的界限和协议,只是为了将基础设施与网络功能隔离开。
- **Or-Vnfm(Orchestrator↔VNF Manager)**:这是 NFV 管理与编排中的接口,定义了资源编排层和虚拟网络功能管理层之间的接口协议,包括资源请求、预留、分配、授权等内容,也包括向虚拟网络功能管理模块下发配置信息和收集状态信息。
- **Vi-Vnfm(Virtualized Infrastructure Manager↔VNF Manager)**:这也是 NFV 管理与编排中的接口,定义了虚拟网络功能管理层和基础设施层之间的协议,负责将虚拟网络管理层的资源请求信息下发到基础设施层并上报硬件资源的配置和状态。
- **Or-Vi(Orchestrator↔Virtualized Infrastructure Manager)**:这也是 NFV 管理与编排中的接口,定义了资源编排层与基础设施管理层之间的协议,完成计算、网络等资源的请求下发并上报虚拟资源和硬件资源的配置和状态。

- **Nf-Vi(NFVI↔Virtualized Infrastructure Manager)**:这是基础设施管理层与基础设施层之间的接口协议,包括向基础设施层请求资源和向管理层上报状态信息等。
- **Ve-Vnfm(VNF/EM↔VNF Manager)**:这是虚拟网络功能管理器与网元/虚拟网络功能之间的接口协议,主要负责网元信息配置和生命周期管理。
- **Os-Ma(OSS/BSS↔NFV Management and Orchestration)**:该协议包括了网元生命周期管理信息、NFV 状态配置信息、管理配置策略以及收集 NFVI 使用量信息。
- **Se-Ma(Service、VNF and Infrastructure Description ↔ NFV Management and Orchestration)**:这一协议接口并未在图 22-18 中体现,但其规定了 VNF 部署模板的下发规范,模板是由管理和编排层根据网络运营商的要求生成的。

22.1.3 网络切片

22.1.3.1 网络切片的定义

所谓网络切片就是一组网络功能、运行这些功能的资源以及这些网络功能特定的配置组成的集合。网络切片将一个物理网络分割成多个虚拟的端到端网络,每个虚拟网络(包括设备、接入网、传输网和核心网等)之间都是逻辑独立的,因此任何一个虚拟网络发生故障都不会影响其他的虚拟网络。

从以上描述我们可以看出,网络切片是为了解决网络功能多样性问题而诞生的技术,是一个逻辑的虚拟网络,也是一个复合层,这个复合层包括了服务层、基础设施层以及它们之间的映射关系。

- **服务层**:从逻辑的角度来描述网络功能之间的联系和组成,这些网络功能通常以软件/进程的方式被定义,也包括了接口等要素。
- **基础设施层**:从物理的角度描述了一个网络切片运行所需要的网络元素和资源,包括了计算资源、网络资源等。
- **两层间的映射关系**:映射关系具有以下两部分内涵:
  - 虚拟功能到物理功能之间的映射:包含了网络转发元素和计算资源的选择,这些所需要的资源的数量和部署方式是由服务层的需求决定的。
  - 虚拟链路到物理链路之间的映射:即分配多少链路带宽给该网络切片,这也是由服务层的需求决定的。

因此,我们可以看出,网络切片是网络从"one size fits all"向"one size per service"的演进,即由过去的一张网络搞定全部服务的形态转变为一个服务一张网络(虚拟网络)的形态,SDN/NFV 则成为了实现网络切片最直接的方式。

如图 22-19 所示,在 SDN/NFV 的网络切片中,子网实例(sub-network instance)可以看作一组相关的 VNF 或 PNF,而网络切片实例(network slice instance)则定义了来自底层资源集合的切片。

在 5G 时代,网络切片是实现业务多样性和业务加速的关键技术,特别是以下三个场景

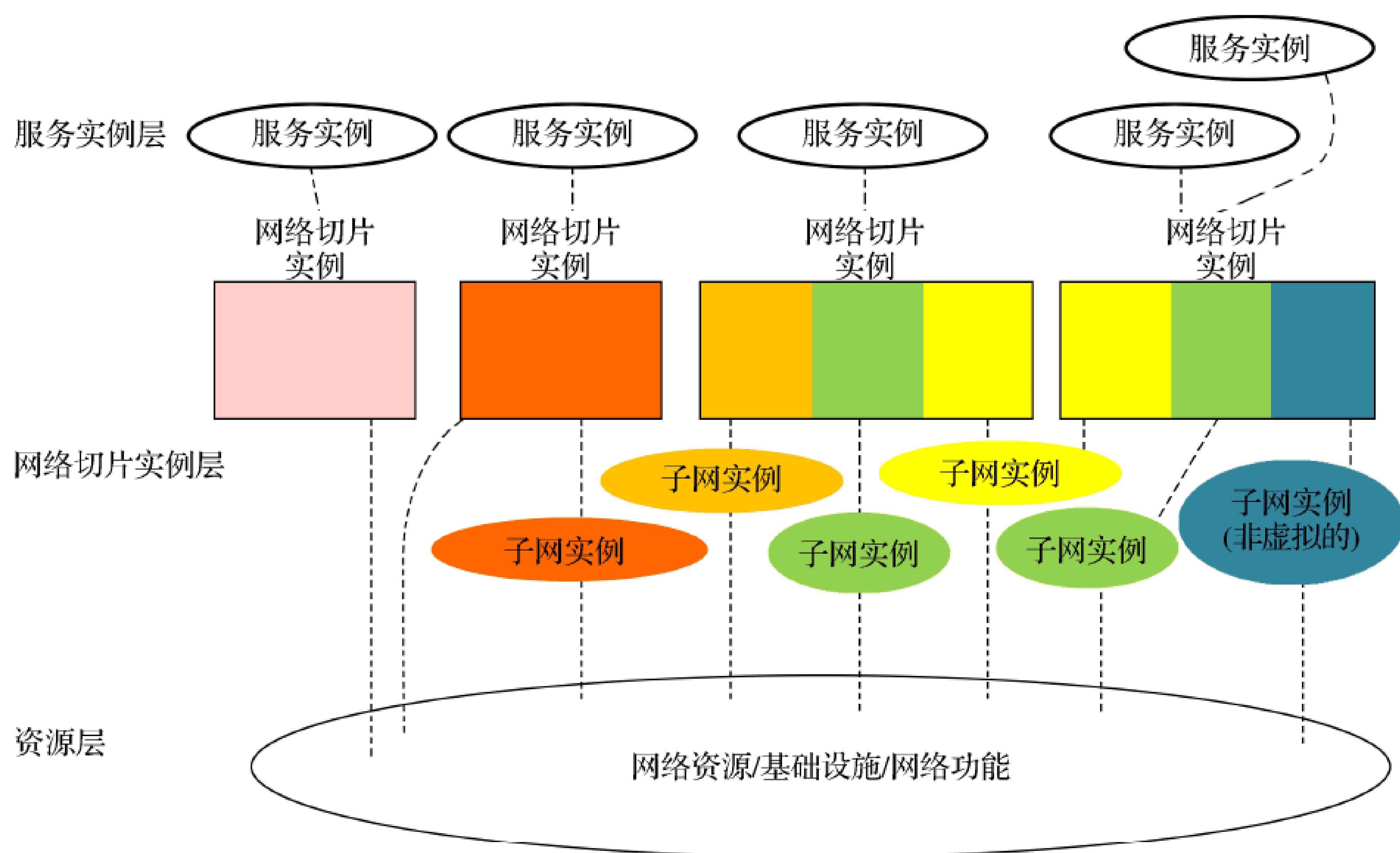


图 22－19　SDN/NFV 中的网络切片

中对于网络切片的要求会越来越高、越来越细致：

- **eMBB(Enhanced Mobile Broadband,增强移动宽带)**：针对的是大流量移动宽带业务,5G 需要以 10 Gbit/s 的数据传输速率支持数万用户。
- **uRLLC(Ultra-Reliable and Low Latency Communication,高可靠低延时通信)**：针对的是诸如无人驾驶等超高可靠性和超低延时性的通信业务。3GPP 对用 eMBB 和 uRLLC 的用户平面和控制平面时延指标进行了描述,要求 eMBB 业务的用户平面时延小于 4 ms,控制平面时延小于 10 ms ;uRLLC 业务的用户平面时延小于 0.5 ms,控制平面时延小于 10 ms。
- **mMTC(Massive Machine Type Communication,海量机器通信)**：针对的是海量物联设备组网传输业务。

在 5G 环境下,每个端到端的网络切片均由无线接入网、传输网和核心网的子切片组成,并通过端到端的切片管理系统进行统一管理。

在 5G 时代,传统网络架构已经不能适应高速而多样的网络服务了,因为不同的业务场景对网络有不同的要求,这些要求之间甚至可能产生冲突。若使用单一的传统网络服务于这些场景,则会要求网络功能面面俱到,这样过于复杂,甚至彼此冲突而无所适从,从而使网络资源低效臃肿。因此整合现有网络基础设施并向上适配多样性的网络服务就成了处于中间层的网络切片的使命,计算机科学领域有一句名言:“任何问题都可以通过引入一个间接层来解决”,网络切片就是解决上述问题的“间接层”。

22.1.3.2　网络切片的分类与部署

网络切片一般分为独立切片和共享切片两类。





➢ **独立切片**：拥有控制平面、用户平面和各种业务功能模块并且具有独立功能的切片。这种切片为特定的用户群体提供独立的端到端的专网服务或特定功能服务。

➢ **共享切片**：可以供各种独立切片共同使用资源的切片。这种切片可以提供端到端的功能，也可以只提供部分共享功能。

这两种切片可以分成三种部署场景。

1）独立切片与共享切片纵向分离的场景

在这种场景下，端到端的控制平面切片作为共享切片，在用户平面形成不同的端到端的独立切片，控制平面共享切片为所有的用户服务，对不同的独立切片统一管理，包括鉴权、移动性管理、数据存储等，如图 22－20 所示。

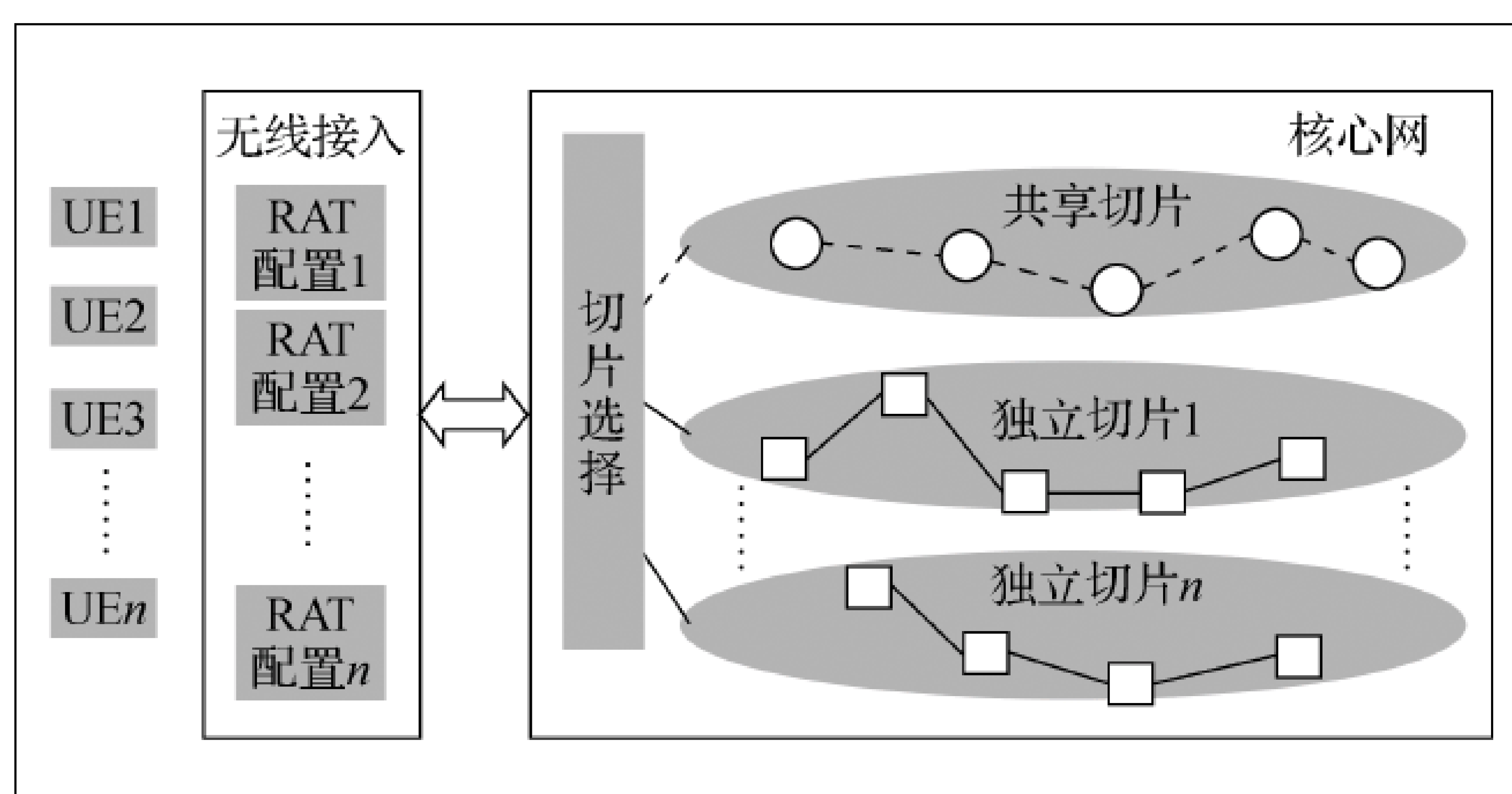


图 22－20　纵向分离场景

2）独立部署各种端到端的独立切片的场景

在这种场景下每个独立切片都包含完整的控制平面＋用户平面的功能，形成了为不同用户群体服务的专有网络，如图 22－21 所示。

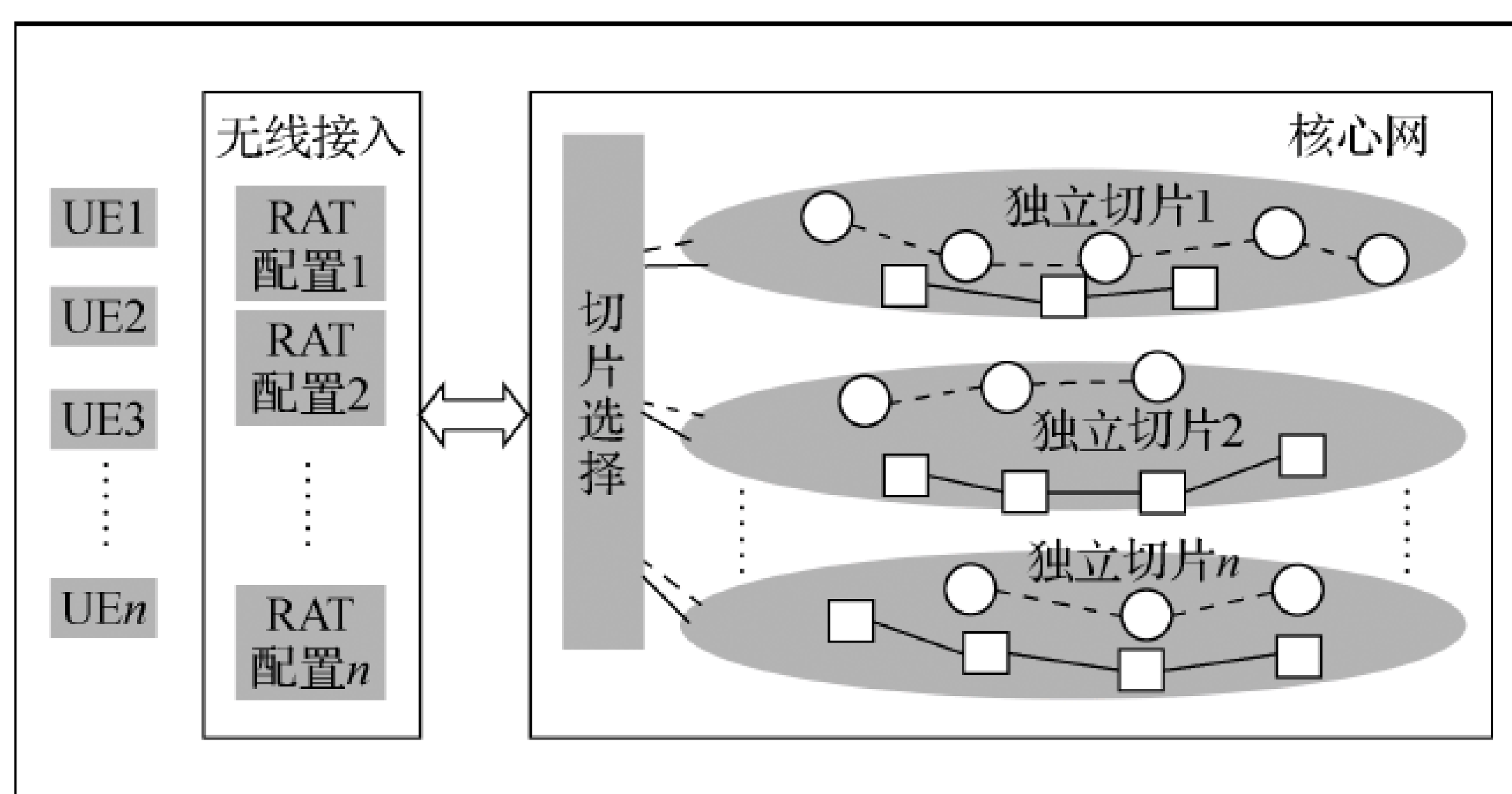


图 22－21　独立部署场景

3）独立切片与共享切片横向分离的场景

在这种场景下，共享切片（通用切片）只实现了一部分非端到端的功能，共享切片后面对接了不同的个性化的独立切片，如图 22－22 所示。

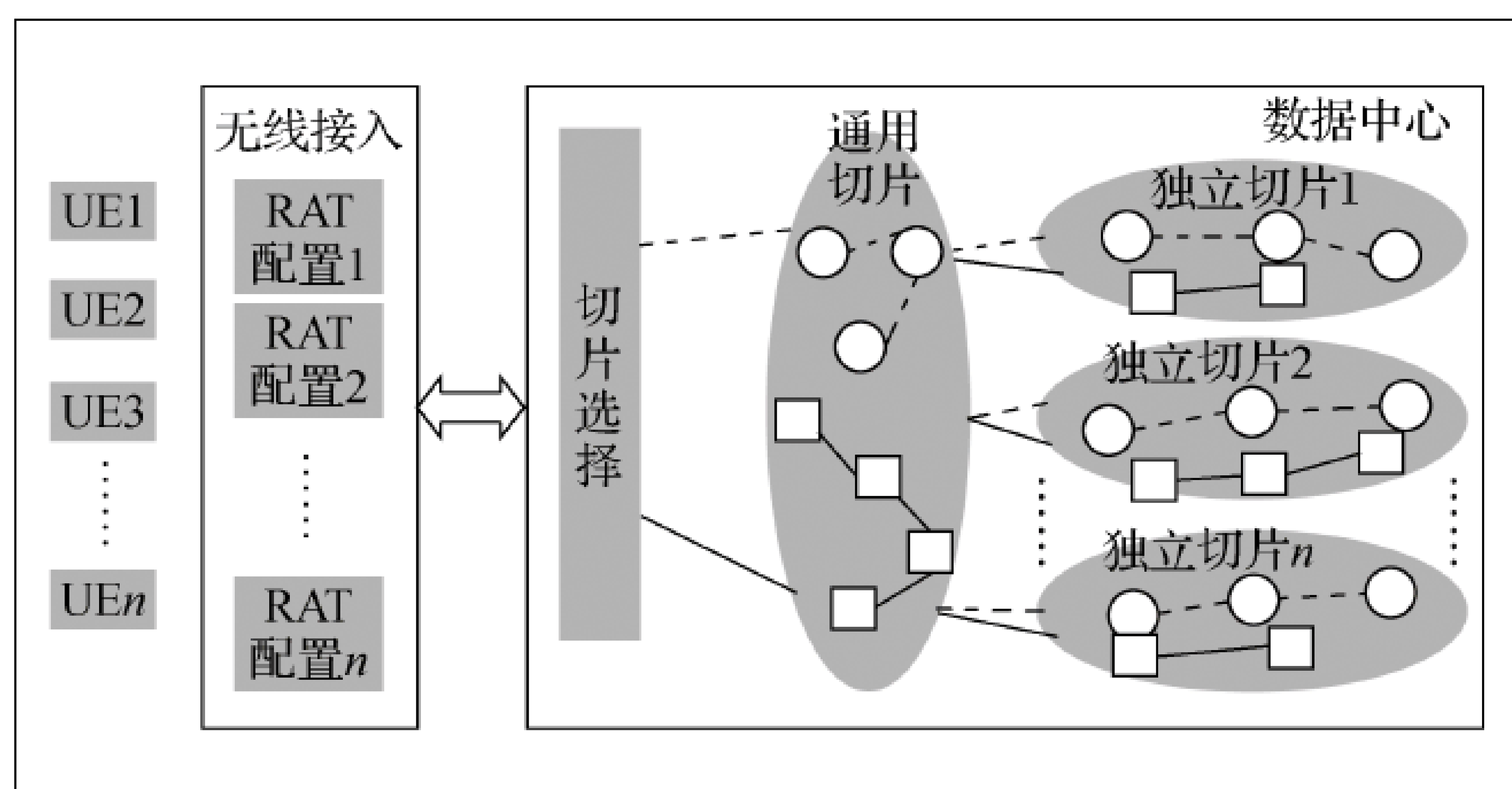


图 22－22　横向分离场景

22.1.3.3　网络切片的生命周期

网络切片资源的生命周期大致可以分为 5 个阶段，这 5 个阶段贯穿了从业务开户到资源注销的全过程：设计阶段→购买阶段→编排阶段→运营阶段→下线阶段，我们在此仅就比较重要的三个阶段进行展开。

1）设计阶段

这个阶段分为切片设计和商业设计两部分。

- **切片设计**：按照规定的业务需求选择相应的特性(功能、时延、安全性等)设计和生成网络切片模板。
- **商业设计**：完成差异化的商业推广和定价规则的制定。

网络切片管理功能的提供方要为需求方提供切片编辑工具和标准化的模板，模板要包括结构与配置的描述、如何进行实例化以及如何控制网络切片实例等要素。

在设计过程中，设计人员选择对应的功能组件，以满足需求方提出的网络业务需求，利用上述工具生成网络拓扑和各部分之间的交互协议，并结合业务特点规划相应的安全性、可靠性和 QoS 要求，以保证资源规模能够匹配性能指标。

总之，设计阶段的核心输出就是网络切片模板。

2）编排阶段

用户购买了网络切片模板后就进入编排阶段，在这个阶段要完成以下任务：

- 配置切片管理器解析模板，并为之后创建网络切片准备必要的逻辑资源。
- 对这些资源进行逻辑隔离，以保证该切片与其他网络切片对应的资源彼此独立。

编排实质上是个全自动化的过程。一个切片模板可以经过多次编排生成多个网络切片的实例。系统会为切片选择和分配最合适的物理/虚拟资源，并完成部署配置和连通性测试。

端到端的网络切片涉及接入网、传输网、核心网等网络，具有较高的管理复杂性。而且各个网络的设备由不同的厂家提供，因此切片的编排、部署和对接都存在一定的难度。

3）运营阶段

运营阶段就是网络切片使用维护的阶段了。网络运营方通过切片管理器提供的接口进

行动态管理,包括:

➢ 对资源、性能等实施监控;

➢ 对网络切片进行升级、维护和调整,支持切片动态伸缩、切片功能增减等;

➢ 根据业务需要对网络切片进行二次开发。

22.2 数据平面加速技术

加速技术的本质就是“让专业的人做专业的事”。随着硬件计算资源性能的提升,特别是 GPU、NPU、ASIC、FPGA 等异构计算体系芯片组件的专业度越来越深,使用这些硬件计算资源做它们最擅长的事可以大大加快计算的进程,节省 CPU 的计算资源。同样,软件模块如果能有效地利用这些术业有专攻的硬件去加速计算进程,也可以达到很好的性能提升效果。

DPDK 和 SPDK 分别作为网络资源和存储资源的软件加速框架,瞄准和利用了计算芯片的一些固有特点和长处,从而使自身的计算性能得到了很大的提升。

22.2.1 DPDK 框架

在 SDN\NFV 的大背景下,网络传输技术领域出现了以下技术趋势:

➢ 使用通用硬件平台(X86/X64)进行网络传输业务,使网络传输的软硬件功能解耦,软件不再绑定特定的硬件,也与处理器架构解耦,这契合了 SDN 的解耦思想。

➢ 利用通用硬件平台将网络数据包解析、发送等固定而重复性的业务“卸载”(offload)到专用硬件平台中进行,以提升传输效率。

- 通用硬件平台意味着啥都能干,但是没有对某一项“术业”有“专攻”,因为通用硬件毕竟要兼顾所有计算场景。
- 利用专业硬件去做大量重复性的专业动作,将原本由软件完成的行为“沉降”到硬件中完成可以大大提升处理的速度。

➢ 处理框架逐渐从内核态转移到用户态。这样做一是降低了开发的门槛,二是提升了系统的鲁棒性(用户态进程崩溃不会产生 BSOD),三是减少了用户态与内核态之间的切换,降低了开销。

面对上述技术趋势,Intel 联合 6Wind 提出了基于 IA(Intel Architecture)平台的 DPDK 框架。DPDK(Data Plane Development Kit,数据平面开发工具包)主要面向 OSI 模型的第二和第三层协议处理,是基于 Intel X86/X64 平台的网络数据包处理框架,用于 Intel 通用平台的网络传输开发领域。但实际上到目前为止,DPDK 不仅仅限于 Intel 的 X86/X64 平台,包括 ARM 和 PowerPC 在内的体系结构都得到了 DPDK 的支持。

DPDK 框架本质上是一套网络数据包旁路处理的方案,所谓旁路,就是“架空”原有的网络协议栈驱动体系,使数据包从另一个纵向协议栈经过,从而进行特定的处理。DPDK 不是

一种新的技术，它是基于"DPDK 所在通用平台只是用于处理网络数据包的收发解析而不用于其他通用型网络业务"这样一种前提的网络加速优化方案，具有以下两种设计思想：

- 基于通用计算平台硬件的各种加速特性（硬件加速）；
- 对操作系统、网络协议栈驱动的软件进行优化改进（软件加速）。

采用 DPDK 框架的旁路化网络协议栈如图 22－23 所示。

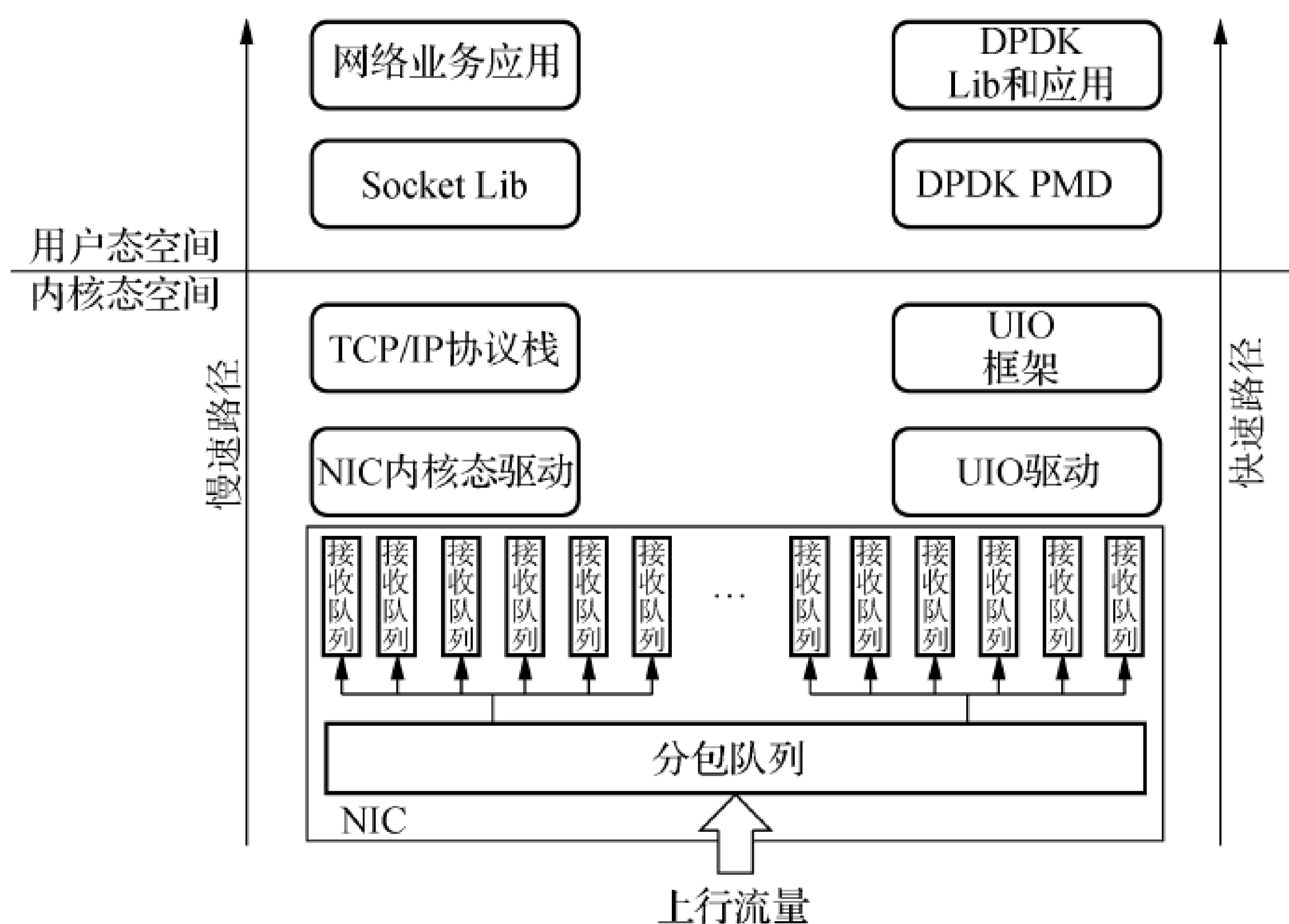


图 22－23　采用 DPDK 框架的旁路化网络协议栈

上述的前提条件限定了 DPDK 框架只能用于网络传输领域，一般用于 SDN 交换机、路由器、网关等网络设备。而这两种设计思想也是对通用型驱动栈和操作系统调度机制的一种破坏，目的是更专心地处理网络数据包，正所谓"不破不立"。当然，这一破一立也要求 DPDK 框架的操作系统是基于 Linux 的，这是因为目前条件下基于 Windows 的平台实现网络协议栈的"架空"远不如 Linux 平台方便。

22.2.1.1　传统的网络传输处理框架

传统的网络数据包的处理是基于中断机制的，我们以收到网络数据包为例来理解其流程，如图 22－24 所示：

（1）网卡收到网络数据包后产生硬件中断。

（2）中断服务例程进行网络数据包的预处理，即执行中断服务例程的前半段。

（3）执行中断服务例程的后半段，在内核处理线程的调度下执行 DPC 处理，将网络数据包向上层（协议栈驱动）传递。

（4）内核中网络协议栈自底向上地解析和处理网络数据包。

（5）通过 AFD（辅助功能驱动）将网络数据包递交到用户态的应用程序缓冲区。

（6）应用程序收到并处理网络数据包。

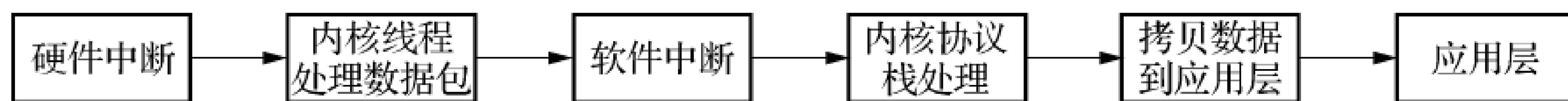


图 22－24　基于中断机制的网络传输处理流程





可以看出，基于中断机制的网络数据包处理是个非常冗长的流程。对于少量的包处理还比较得心应手，如果网络数据包的并发量很大，那么源源不断的中断会使系统处于上下文频繁切换的巨大性能开销中。中断，英文为 Interrupt，也可以被解释为"打扰"。试想一个心无旁骛专心工作的人被喋喋不休的外部事件打扰该是多么崩溃的一件事情啊！

由此可以看出，传统网络处理框架在处理高速、大量的网络数据包时具有以下弊端：

- 中断机制使线程上下文切换开销达到最大。
- 处理流程过于冗长，破坏了网络数据包处理的高速性、实时性。
- 对于大量的处理步骤相同的网络数据包，协议栈驱动处理的步骤过于繁杂，许多步骤都是不必要的，进一步降低了处理的时效性。
- 存在内核态内存空间向用户态内存空间拷贝的巨大开销。
- 内核工作在多核系统中，必须考虑全局一致性问题。但即使采用了 Lock Free 这样的机制也避免不了锁总线、内存屏障等带来的性能损耗。
- 在多路 CPU 平台中，中断处理可能跨越多个 CPU（例如中断发生在 CPU0，ISR 却运行在 CPU1），这样做可能导致缓存命中失效，从而带来了性能损耗。

22.2.1.2　DPDK 框架关键技术

DPDK 采用旁路化处理机制绕开了固有的内核网络协议栈，其流程也做了简化。应该说 DPDK 的通用性不强，因为 DPDK 严格来讲只是个数据包处理框架，I/O 的单位都是"帧"（Frame），需要上层各种协议栈 I/O 模型的配合，例如 TCP 组帧、分包、发送窗口计算等功能它都是没有的，因此软件生态远不如现有操作系统的协议栈丰富（例如前文中介绍的 Windows 下的 IOCP 机制、Linux 下的事件多路分离机制以及更上一层的用户态的 ACE 框架等就不能使用了，腾讯和阿里针对 DPDK 都开发了自己的协议栈，如 F-Stack 和 Ali-Stack，以支持 socket 机制和原有的模型与框架）。但从另一个角度来说，DPDK 框架本来就是用于基于通用平台的交换机和路由器等网络设备的，其场景的专有性和细分性也使上述问题不再成为问题。

对比基于中断的机制，DPDK 框架的处理流程做了很大的调整，如图 22－5 所示：

（1）网卡收到网络数据包后产生硬件中断。

（2）由于 DPDK 接管了原有的协议栈处理流程，因此中断服务例程被跳过，放弃执行。

（3）由于应用层实现已经建立了从用户态缓冲区到内核态网络数据包缓冲区的映射，因此用户态的应用层通过直接映射机制和主动查询方式获取网络数据包。

（4）应用进程对网络数据包进行处理。

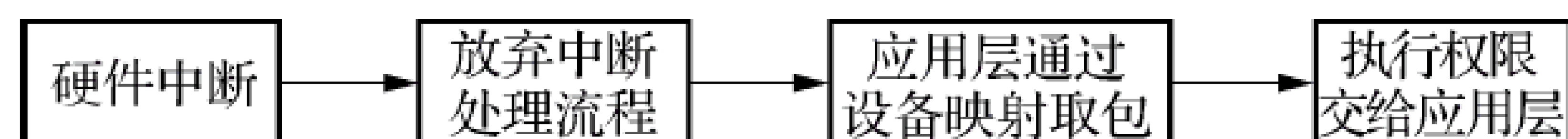


图 22－25　基于 DPDK 框架的网络传输处理流程

可见针对基于中断机制的处理流程中的弊端，DPDK 框架做了以下改进：

- 摒弃了中断机制，使用轮询机制主动查询网络数据包，抵消了中断切换的系统开销；

➢ DPDK 框架大大简化了网络协议栈的处理流程；
➢ 使用 UIO 机制，最大限度地减少了跨用户态空间的内存拷贝；
➢ 使用 CPU 亲和性消除了跨 CPU 处理的切换开销；
➢ DPDK 框架提供了多种功能库，使 DPDK 不仅仅定位为网络协议栈，也成了网络 I/O 开发框架。
  - 可以基于 DPDK 框架实现网络报文的标准流水线模型。
  - DPDK 框架提供了包括 librte_port、librte_table 和 librte_pipeline 在内的功能库，定义了处理网络数据包的标准方法。

前文说过，DPDK 框架是对网络协议栈的旁路替代和颠覆，但其本身并没有什么新颖的技术，因此可以认为 DPDK 是一个 I/O 优化方案，是多种优化技术的组合。这些技术包括：

1. UIO 技术

Linux UIO（User space I/O，用户空间 I/O）技术是运行在用户态空间的 I/O 技术。Linux 系统中常规的设备驱动都是运行在内核态空间的，用户态进程使用系统调用 API 与驱动交互。UIO 技术是将小部分驱动运行在内核态空间（硬中断只能在内核态空间处理），大部分运行在用户态空间，用户态进程无需跨空间即可访问驱动程序。

但 UIO 不是说驱动"全部都"在用户态空间，在内核态空间中可能也存在一部分功能模块，这部分内核模块一般要完成两项工作：

➢ 分配记录设备需要的资源，并且注册 UIO 设备驱动。注册是通过调用 Linux 的 UIO 注册 API（uio_register_device 方法）实现的。注册后设备的内核态地址空间会被 mmap 方法映射到用户态内存空间中，以方便用户态进程直接操纵设备内存空间。
➢ 实现一小部分中断服务例程。

而在用户态空间中的那部分驱动程序的职责一般是通过 read、poll 等方法获取底层中断事件，并处理读写数据等具体事务。

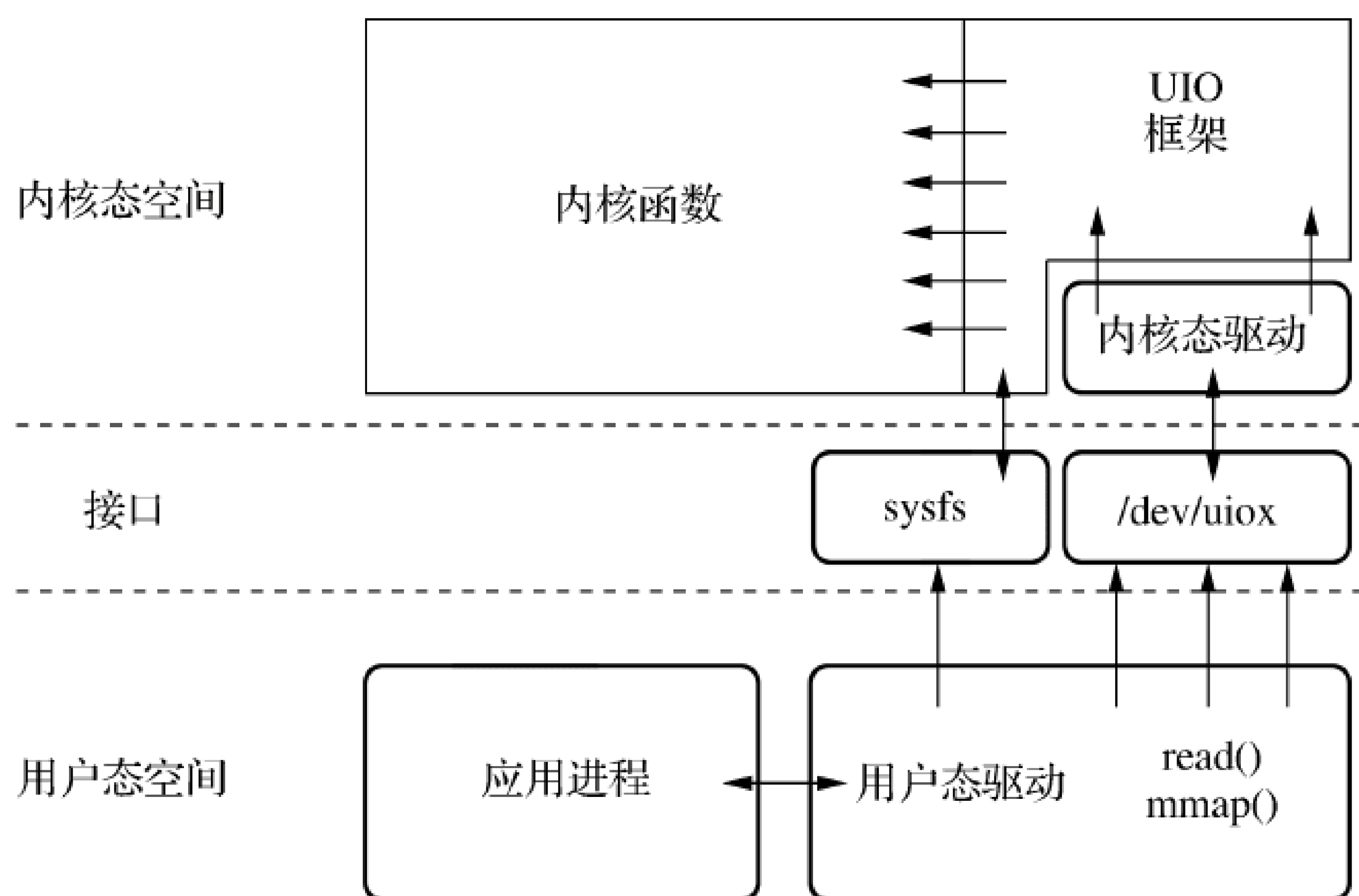


图 22－26　UIO 框架与机制

正是基于UIO技术，DPDK框架实现了内核协议栈驱动的旁路化。本质上UIO技术是通过向用户态空间的进程暴露文件接口实现的，例如在图22－26中对于/dev/uiox的读写就是直接对设备内存的读写。

2. SIMD机制

SIMD即单指令多数据流（Single Instruction Multiple Data），DPDK框架基于向量式编程，采用批量方式可在一个周期内同时处理多个网络数据包。

3. 内存池技术

DPDK框架所使用的内存池指的是用户态空间的内存池，或者说内核态空间和用户态空间的内存交换不是通过内存拷贝方式，而是只做控制权转移。DPDK使用内存池存放代表网络数据帧的Mbuf结构。

DPDK将网络数据帧封装在Mbuf数据结构中，当然Mbuf也可以用于封装通用的控制信息。Mbuf的头部占用两条Cache line（Cache line对齐有利于头部数据载入缓存），第一条Cache line存放包处理中的基础性的、需要频繁访问的信息，第二条Cache line存放一些扩展信息。对网络数据帧封装和处理时，如果数据量少则可以采用一个Mbuf存放，如图22－27所示；而面对巨型帧，单个Mbuf是放不下的，此时可以采用Mbuf链表的方式存放（Mbuf有一个指向下一个Mbuf结构的指针，可构成链表），如图22－28所示。

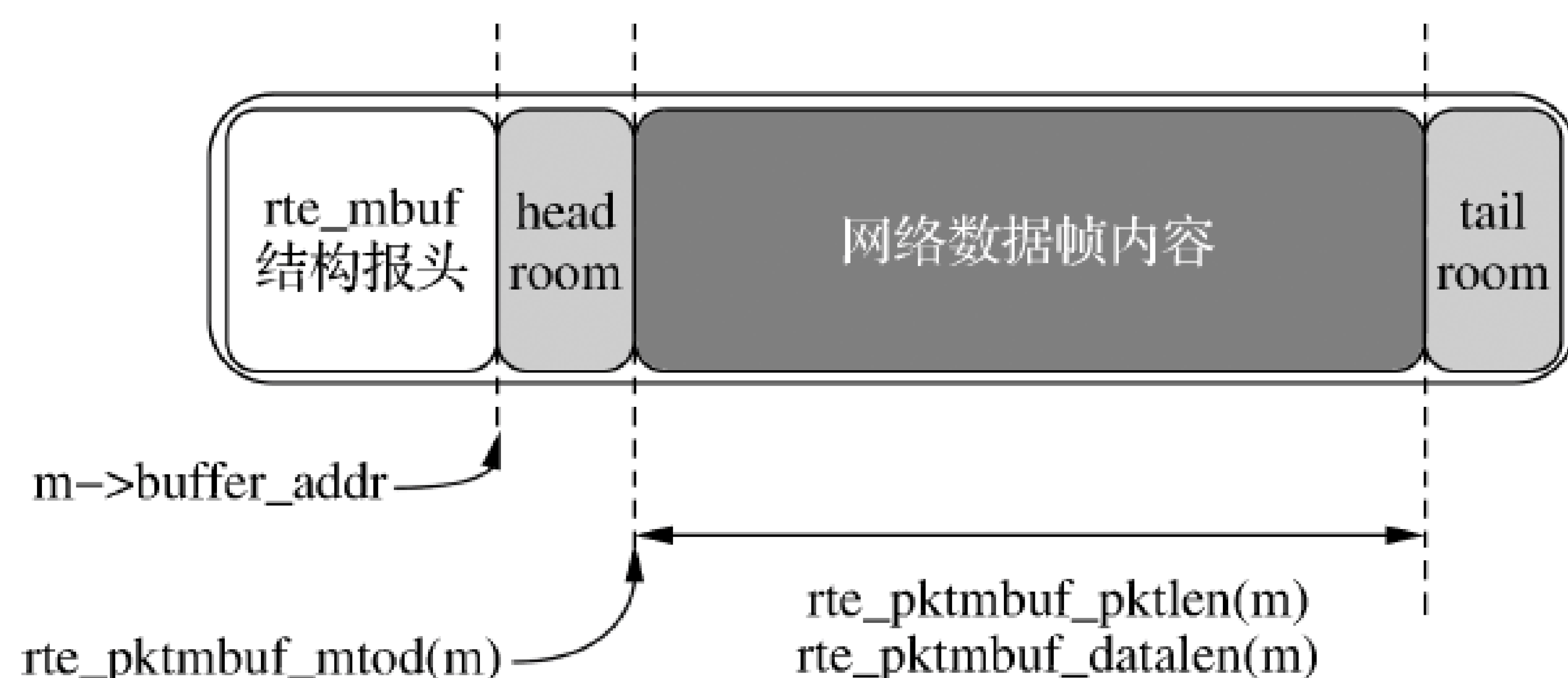


图22－27　单帧的Mbuf结构

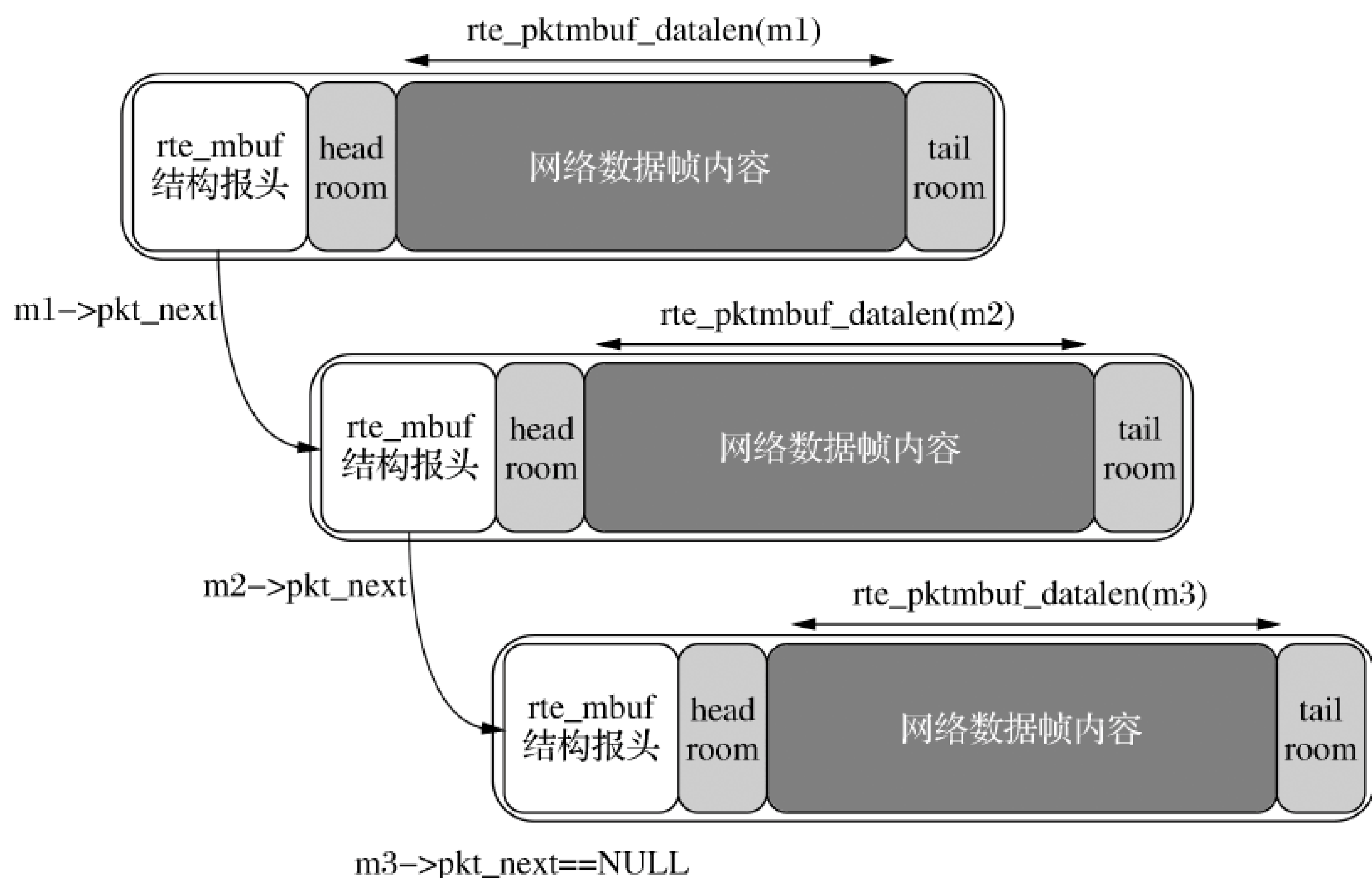


图22－28　巨型帧的Mbuf结构

rte_mbuf 结构对象存放在内存池中，内存池使用环形缓冲区来保存空闲对象。head room 用来存储和系统中其他实体交互的信息，如控制信息、帧内容、事件等。巨型帧的元数据仅出现在第一个 Mbuf 结构中。网络帧元数据的一部分内容是由 DPDK 的网卡驱动写入的。

我们在封装和处理网络帧时，既可以将网络帧元数据和帧本身的内容存放在固定大小的同一段缓存中，也可以将元数据和帧本身的内容存放于两段缓存中。显而易见，前者注重访问效率（只访问一次缓存即可获得全部数据）；后者注重缓存使用效率（数据帧大小可以自定义）。为了访问效率，DPDK 选择了前者。

内存池使用环形缓冲区保存 Mbuf 数据结构，实际上在内存池初始化的时候也同时生成了许多空的 Mbuf 结构，这样在后面使用时就不需要耗费时间创建了。

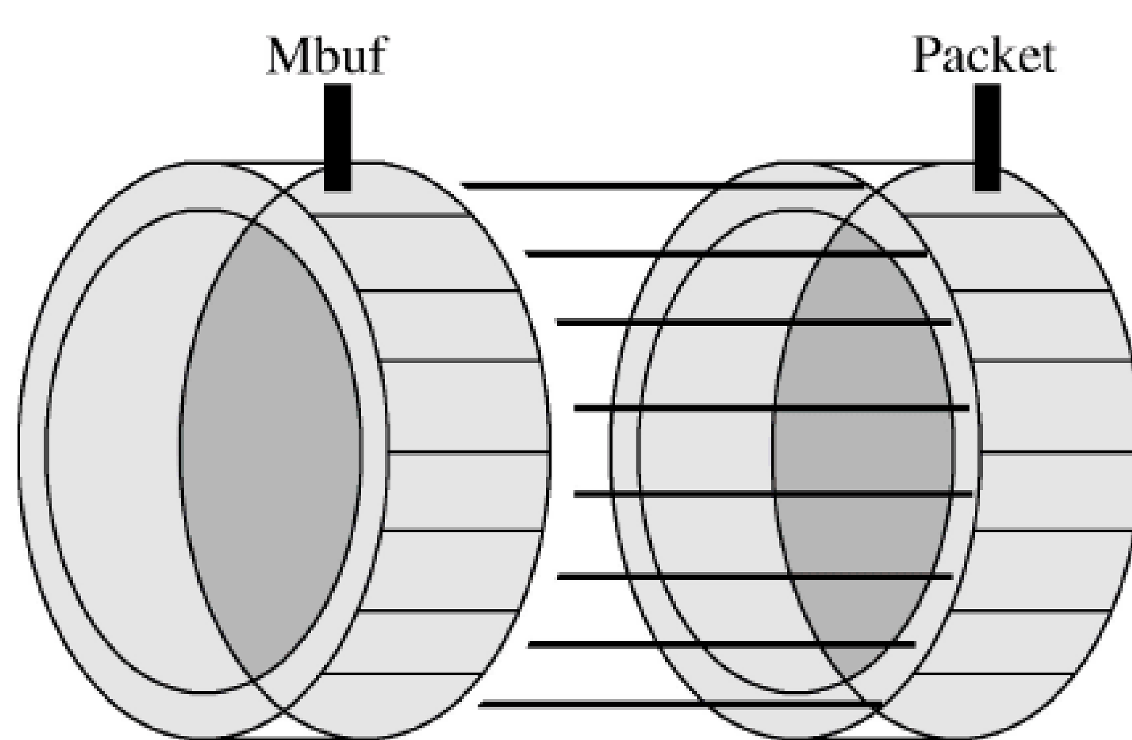


图 22－29　双环形缓冲区结构

这里要注意的是，对于一个网络数据帧，它自身的帧内容和代表它的 Mbuf 结构是存储在两个不同的环形缓冲区里的，当然帧内容会与 Mbuf 一一对应地关联起来，如图 22－29 所示。一般情况下对帧的操作就是对 Mbuf 的操作，只有在需要的时候才会访问实际存放网络帧的缓冲区，这进一步降低了内存拷贝的开销。

4. 大页内存机制

在 X86/X64 系统下，内存页大小为 4 KB。在面对大并发量的网络数据包处理时，内存页的倒换还是很频繁的，因此使用大页内存（Huge Page）机制（例如设置 4 MB 或 1 GB 的内存页）甚至巨页内存机制可以有效地减少内存页的倒换，并有效地提高缓存的命中率，对报文的转发性能影响很大。使用大页内存就是为了使程序尽量独占内存以防止内存换出，扩大页表的命中率。

5. PDM 机制

DPDK 网卡驱动完全抛弃中断模式而改为基于轮询方式收包，避免了中断开销，这种机制简称 PMD（Pool Mode Driver，池模式驱动）。但由于运行在 PMD 模式下的处理器核心会处于 100% 满负荷的状态，因此网络空闲时处理器长期空转，会带来能耗问题。

DPDK 改进了 PMD 机制，推出 Interrupt DPDK 模式，即没有网络数据包的时候处理器中的线程进入睡眠状态，此时改为中断通知方式，且可以与其他非 DPDK 线程共享同一个处理器核，但是 DPDK 线程拥有更高的调度优先级。只有当大量网络数据包源源不断地到来时，处理器才启动 PMD 机制。

PMD 虽是基于 UIO 技术的，但其内核部分负责将网卡 PCI 资源映射到用户态空间中；用户态部分负责轮询网卡 PCI 配置空间的内容以进行网络数据包收取，用户态驱动也可以通过网卡寄存器的状态来判断是否有数据包到达。总体来看，PMD 的特点就是用户态（为主）、轮询和零拷贝。当然也不是所有场景都适合 PMD，只有在 I/O 设备性能远好于处理器





性能且 I/O 吞吐量非常大的情况下才合适，反之可能中断模式更有效率。

6. 无锁化的循环队列机制

DPDK 框架基于 Linux 内核的无锁化环形缓冲队列机制，支持单生产者入队–单消费者出队和多生产者入队–多消费者出队的操作模式，在传输时可以提高传输效率并保证数据同步，如图 22 – 30 所示。

环形缓冲区一般有一个读指针和一个写指针，分别指向队列中的可读数据和可写数据。当存在多个读写用户访问环形队列的时候，应该添加互斥机制来保护访问的并发性。在 Linux 内核中，kfifo 就是一种无锁化的环形队列结构，kfifo 的英文全称就是"Kernel First In First Out"，它采用了并行无锁技术，实现了不加锁就可同步多生产者—多消费者模式的共享队列。

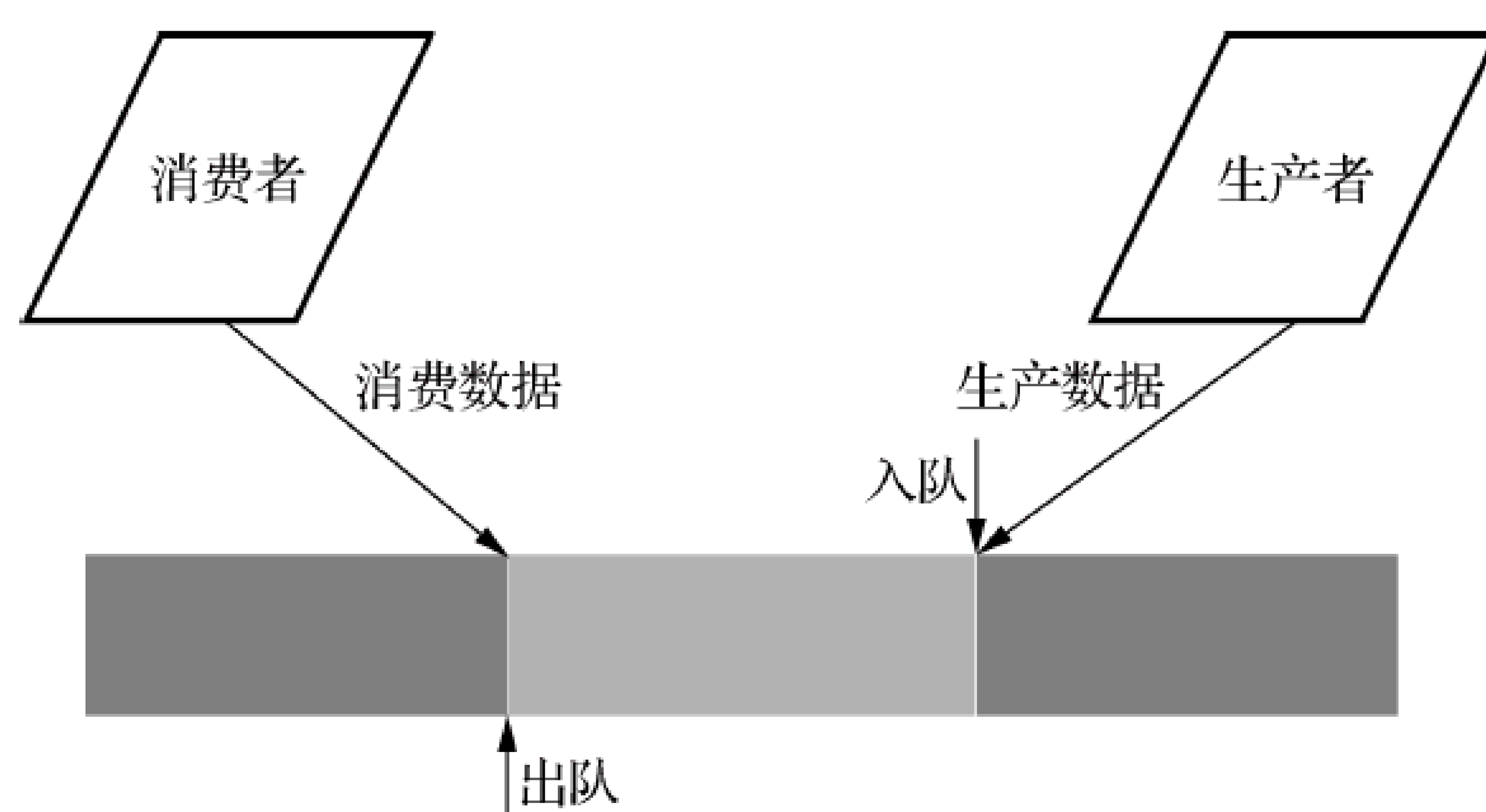


图 22 – 30　kfifo 数据结构

7. 处理器亲和性机制

利用处理器亲和性(CPU Affinity)机制将 I/O 线程绑定到若干个 CPU 核上，以此减少线程调度和切换的开销。同时由于线程被绑定在固定的 CPU 核上，因此 CPU 缓存的命中率大大提高，不会出现前文所描述的中断发生与中断处理跨 CPU 的情况。

CPU 亲和性机制包含两种策略：**Soft Affinity**(**软亲和**)和 **Hard Affinity**(**硬亲和**)。

- Soft Affinity 仅是一个建议机制，如果不可避免，调度器还是会把本线程调度到其他 CPU 上。
- Hard Affinity 是一种强制机制，是调度器必须遵守的规则，无论是否可以避免，都不能将本线程调度到其他 CPU 上。

8. 多队列机制

多队列网卡技术最初是用来解决网络 I/O 的 QoS 问题的。后来随着网络带宽的不断提升，单核 CPU 已不能满足网卡的需求，通过多队列网卡驱动的支持，将各个队列所绑定到不同的 CPU 核上，以满足网卡高吞吐量的需要。

DPDK 将网卡的收发队列与 CPU 核绑定，也就是说该队列收到的报文都交给队列所绑定核上的 DPDK 报文处理线程处理。可以采用两种方式将网络数据包投递到指定的队列中：根据网络数据包五元组的散列值投递到某个队列；通过查找 Flow Director 表项将网络数据包投递到某个队列。前者被称为接收端扩展(Receive Side Scaling，RSS)机制，这是由微软

提出的;后者被称为包字段精确匹配机制,也叫作 Flow Director,是由 Intel 提出的。RSS 是基于网卡硬件的技术,散列值的计算必须在硬件芯片中完成(软件的计算开销很高)。Flow Director 也是基于网卡硬件的技术,且目前仅限于 Intel 的网卡,通过在网卡中设置的 Flow Director 表(类似 SDN 交换机中的流表)来指引网络包的后续处理步骤。驱动程序负责 Flow Director 表的维护,可以对不同类型的包指定不同的 Flow Director 关键字。图 22 - 31 展示了这两种机制的比较。

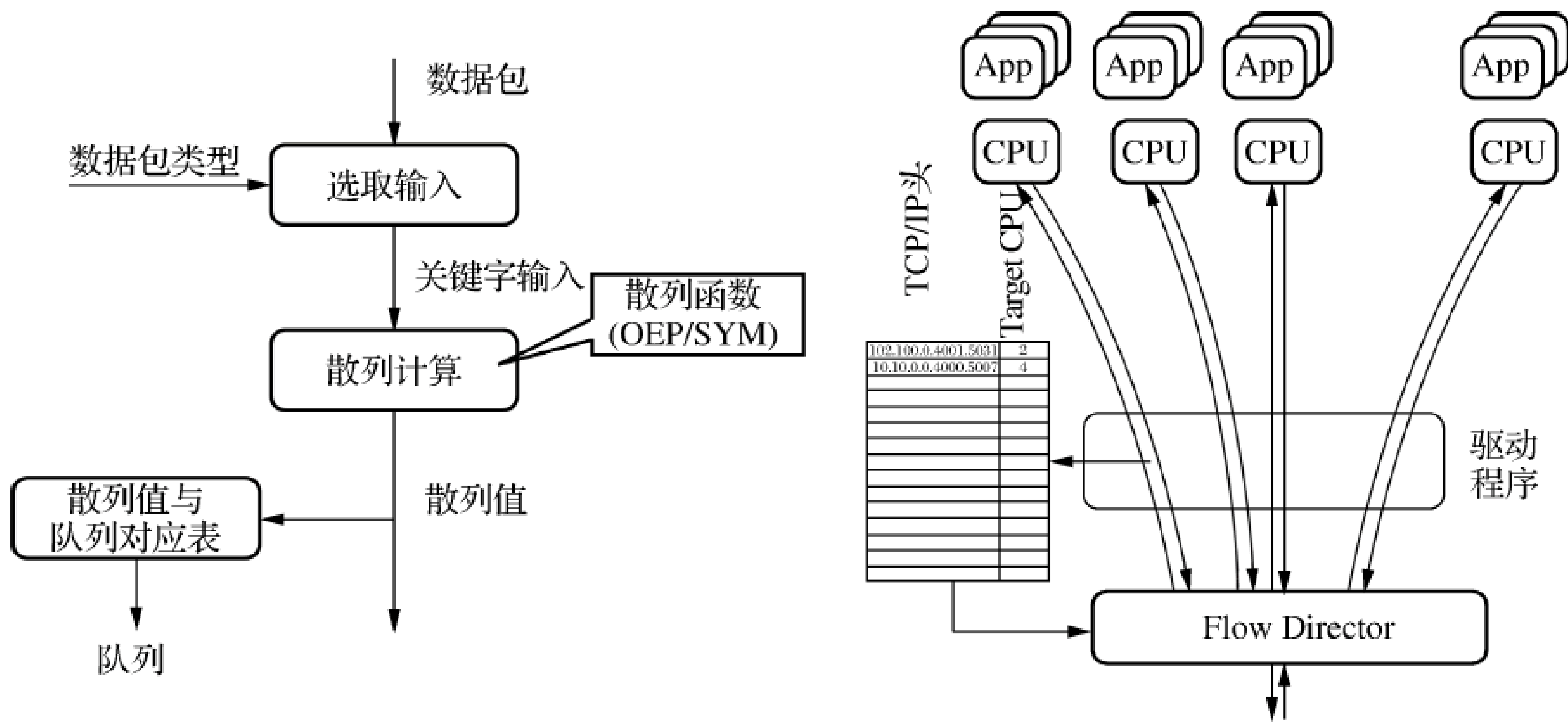


图 22 - 31　RSS 机制与 Flow Director 机制的比较

这两种方式可以结合使用,也可以仅使用一种。例如对于网络数据报文,可以采用 RSS 机制散列到不同的处理器核上进行转发,如图 22 - 32 所示;而对于网络控制报文,可以采用 Flow Director 机制路由到专用的处理器核上处理和转发。

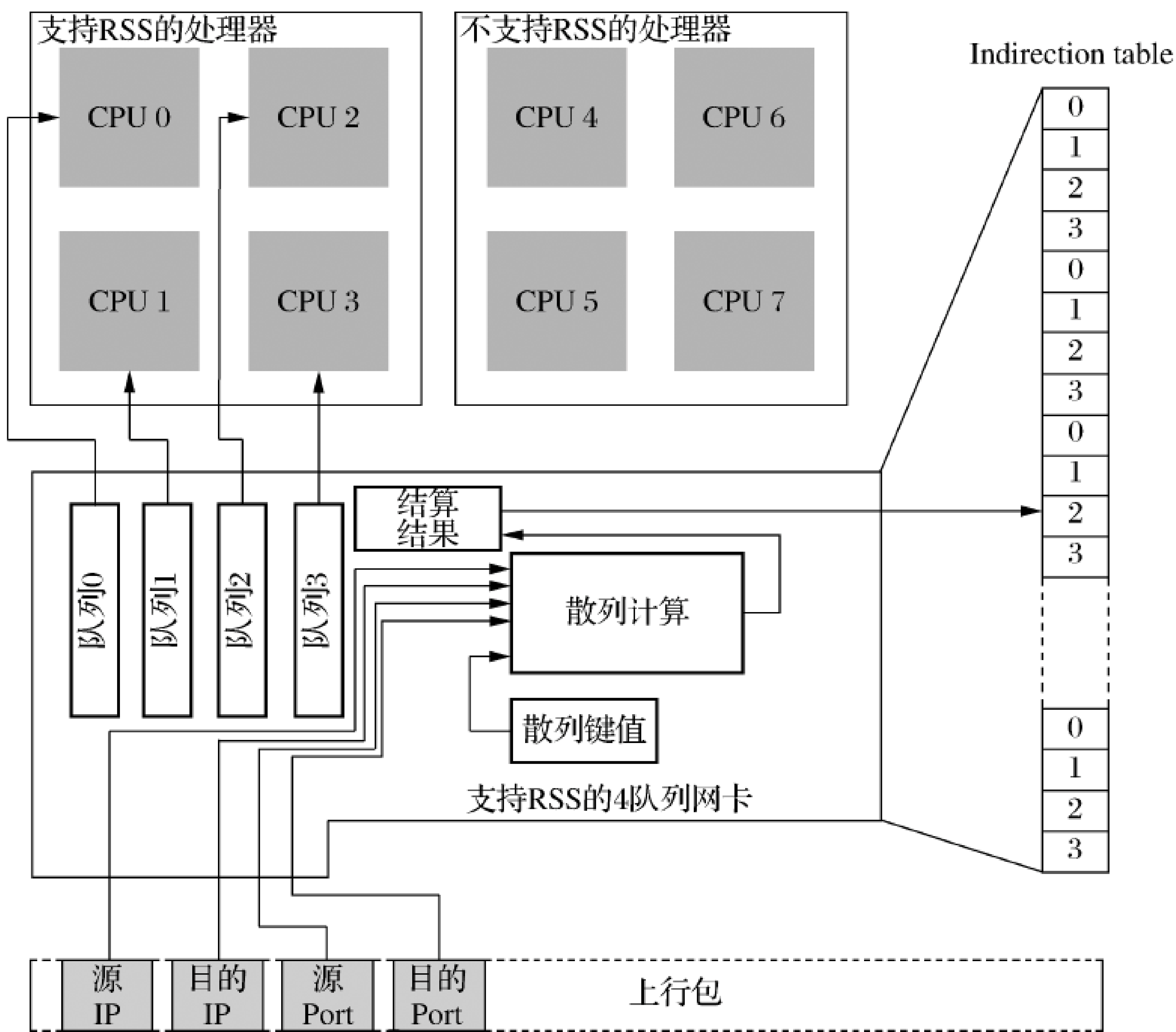


图 22 - 32　RSS 方式逻辑视图

9. DDIO 技术

DDIO(Data Direct I/O,数据直接 I/O)技术是 Intel 提出的一种技术,只能应用于 Intel 的 CPU 上(Intel Xeon E5 及以上的处理器)。DDIO 是一种能够使服务器更快处理网络包的技术,可以有效提高 I/O 吞吐率并降低延迟。其本质就是使网卡缓存与 CPU 通过 LLC(Last Level Cache,三级缓存)直接交换网络数据,从而绕过主存,既缩短了交互的流程,也提升了交互的速度,如图 22-33 所示。当然这要求缓存具有较大的容量,因此 DDIO 技术一般常见于三级缓存而罕见于一、二级缓存。

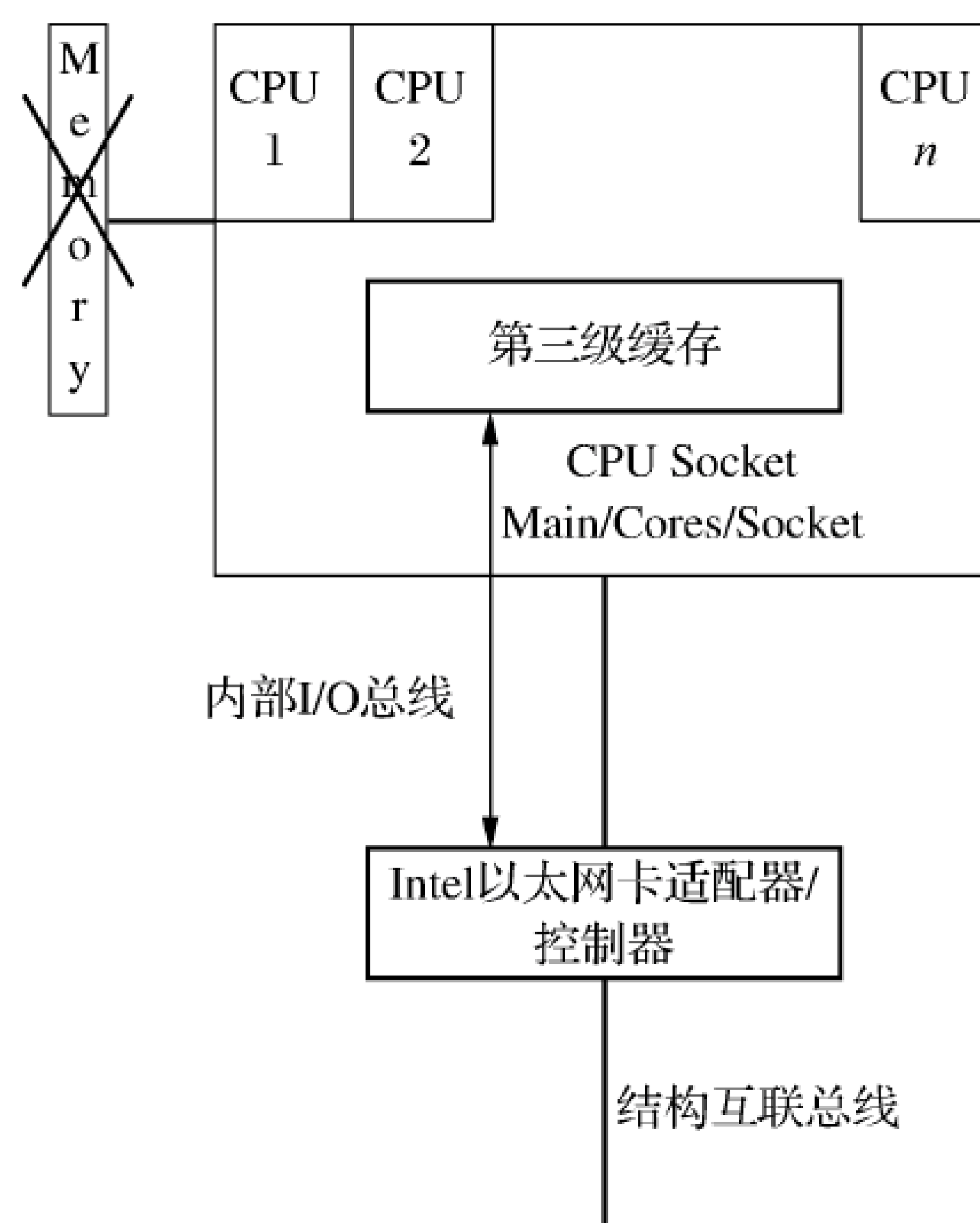


图 22-33 基于 DDIO 的网络数据交互

下面我们根据网卡发送网络包和收到网络包两种情况来感受 DDIO 技术带来的效率提升。

1) 网络数据包发送流程

(1) 不采用 DDIO 技术

在网络数据包发送时,CPU 需要将待发送的报文写到网卡缓存中。因此首先要将缓存中的报文“热数据”写到内存中,还需要通知网卡进行读取操作(站在网卡的角度,要发送的数据在内存中,因此需要从内存中“读取”要发送的内容到网卡自己的缓存中)。网卡从主存读取了数据包并拷贝到网卡内部的缓存,最后再发送到网络上。可见整个过程涉及两次内存读写。

(2) 采用 DDIO 技术

在网络数据包发送时,直接将尚在 LLC 中的报文“热数据”写到 PCI 总线上,继而写到网卡缓存中,直接绕过了主存的读写。

两种方式下的网络数据包发送流程如图 22-34 所示。

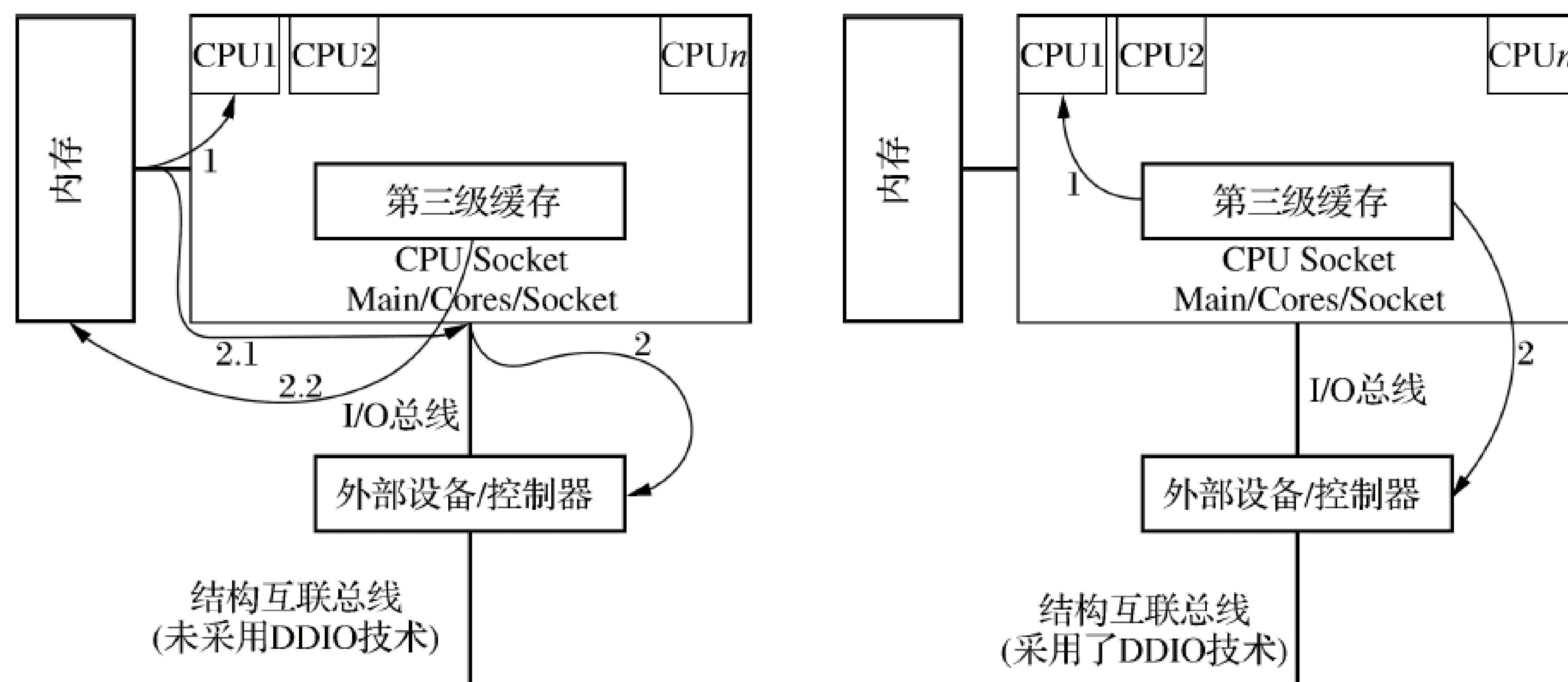
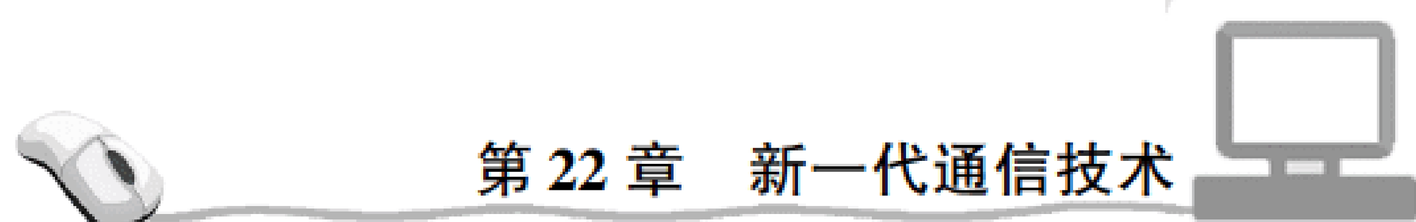


图 22-34 两种方式下的网络数据包发送流程



2）网络数据包接收流程

（1）不采用 DDIO 技术

当网卡收到网络数据包时，将收到的内容投送到 PCI 总线上，继而写到内存中。如果 Cache 中存在这块内存的 Cache line，则需要将 Cache 中的内容先写回到主存中（如果 Cache 中是脏数据），再写回到磁盘，然后新的网络包才能被写到主存中。

写入完成后网卡驱动通知接收线程：一个新的网络报文到达，CPU 探测缓存是否命中，不命中则从内存中访问该网络报文。

（2）采用 DDIO 技术

当网卡收到网络数据包时直接将收到的内容投递到 PCI 总线上，继而直接写到 LLC 中，从而绕过了主存的读写。如果这块缓存区域恰好可用，则直接覆盖更新 Cache 中的内容，并通知 CPU；若不可用，则首先在 LLC Cache 中分配一块区域并更新 Cache 表，以表明该内容对应于内存中的某个区域。

两种方式下的网络数据包接收流程如图 22－35 所示。

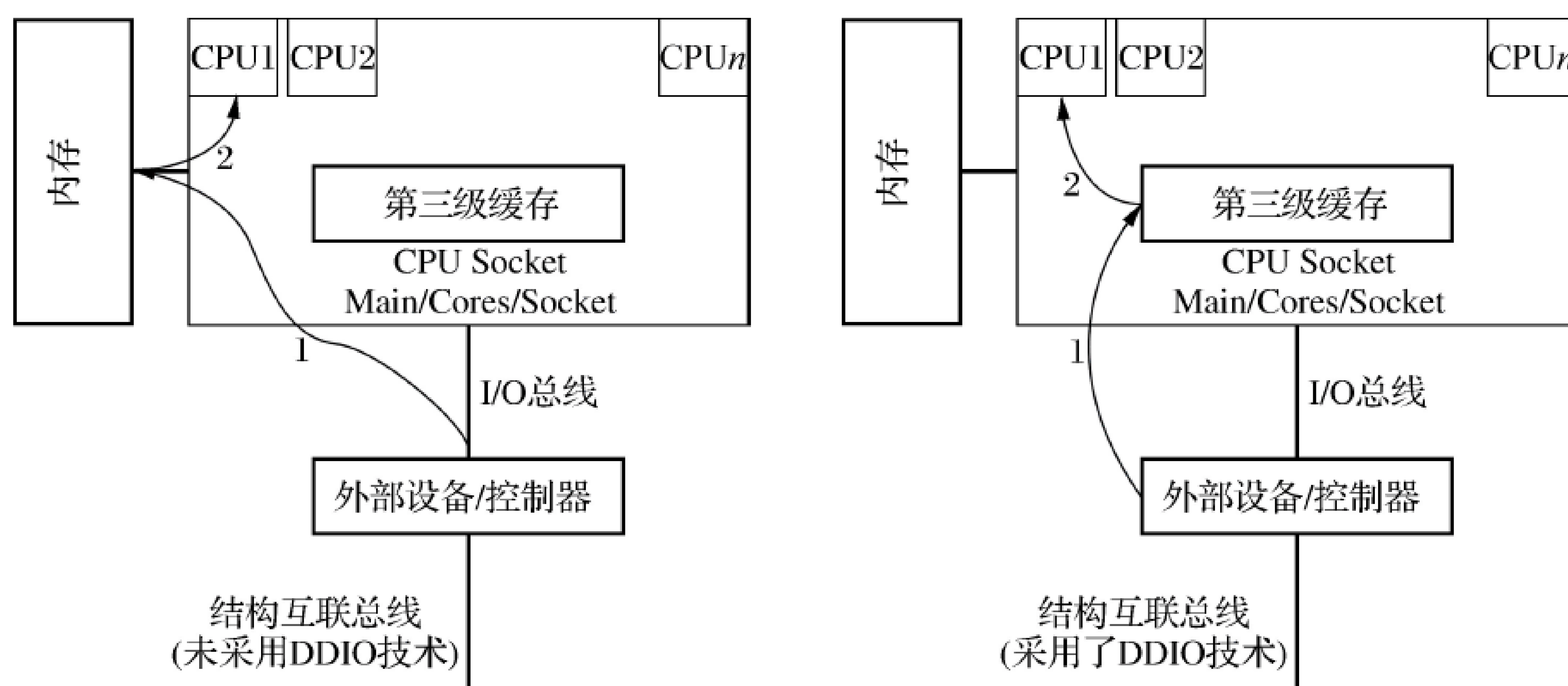


图 22－35　两种方式下的网络数据包接收流程

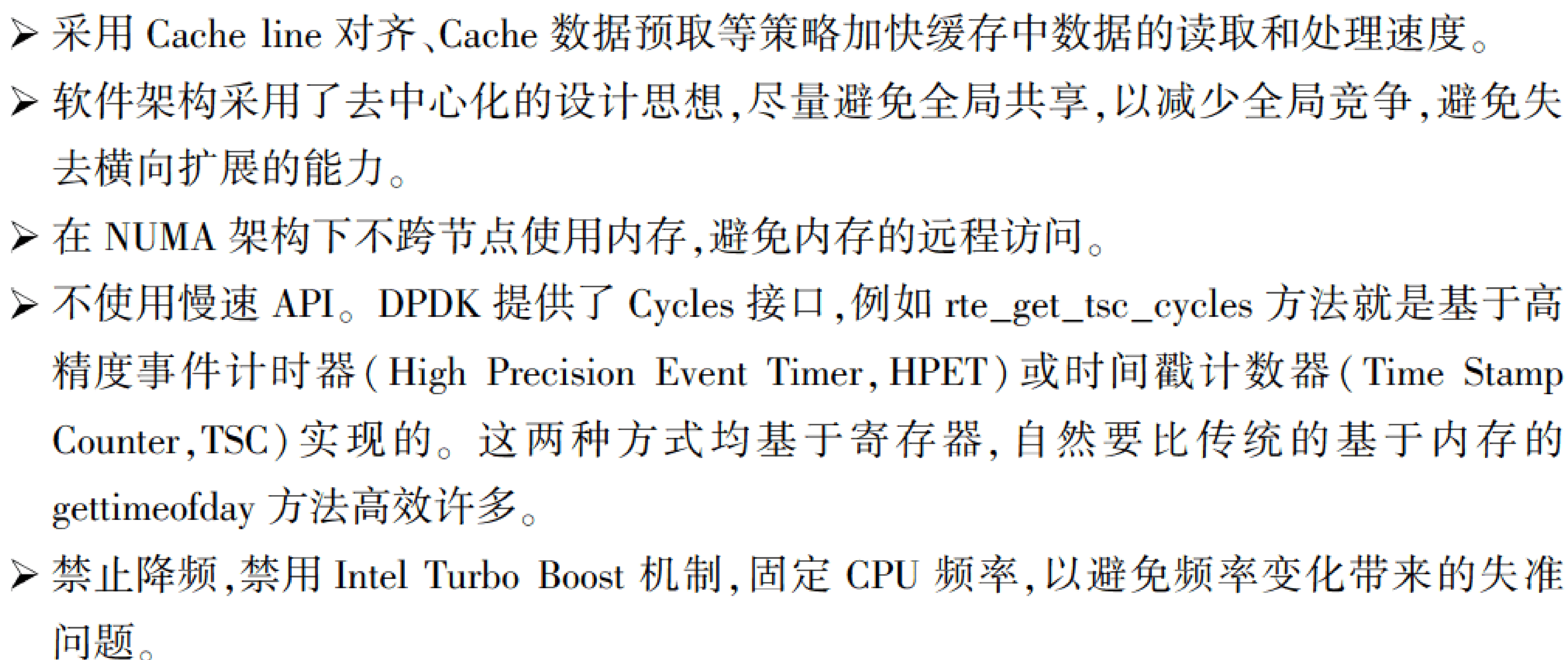
10．硬件加速技术

通过将基础性、重复性的软件工作“卸载”给硬件去完成是提高执行效率极为有效的手段。在 DPDK 框架中，硬件加速的含义是将原本由软件实现的功能“卸载”给网卡，例如计算分析类任务、TCP 组包类任务和 TCP 分段类任务等。

- **计算分析类任务**：包括 VLAN 的硬件卸载，例如 VLAN Tag 的识别、过滤、剥离、插入和对多层 VLAN 的处理；精准事件同步协议中对于时间戳的支持；三层和四层网络包中对于校验和（Checksum）的校验支持等。
- **TCP 组包类任务**：支持对 TCP 收包时的小片段组合，以便于向上层驱动回调较大的数据片段。
- **TCP 分段类任务**：支持对 TCP 大片段发送时的分片处理。

11．其他机制

为了实现 I/O 高效率，在 DPDK 框架中也采用了软件调优的思想：



- 采用 Cache line 对齐、Cache 数据预取等策略加快缓存中数据的读取和处理速度。
- 软件架构采用了去中心化的设计思想，尽量避免全局共享，以减少全局竞争，避免失去横向扩展的能力。
- 在 NUMA 架构下不跨节点使用内存，避免内存的远程访问。
- 不使用慢速 API。DPDK 提供了 Cycles 接口，例如 rte_get_tsc_cycles 方法就是基于高精度事件计时器(High Precision Event Timer, HPET)或时间戳计数器(Time Stamp Counter, TSC)实现的。这两种方式均基于寄存器，自然要比传统的基于内存的 gettimeofday 方法高效许多。
- 禁止降频，禁用 Intel Turbo Boost 机制，固定 CPU 频率，以避免频率变化带来的失准问题。

可以说，DPDK 框架的高效率来源于各种加速机制的“组合拳”，这些拳法中的每一种都不是什么新技术，但是联合起来使用就会产生神奇的效果，因此我们说 DPDK 框架就是一整套优化方案的组合。

22.2.1.3　DPDK 框架的结构和工作流程

DPDK 框架的主要模块如图 22－36 所示。这些模块以基础软件库的形式为上层应用提供基础开发组件，同时这些库也充分利用了上文中所描述的软硬件加速技术，为高速处理网络数据包赋能。

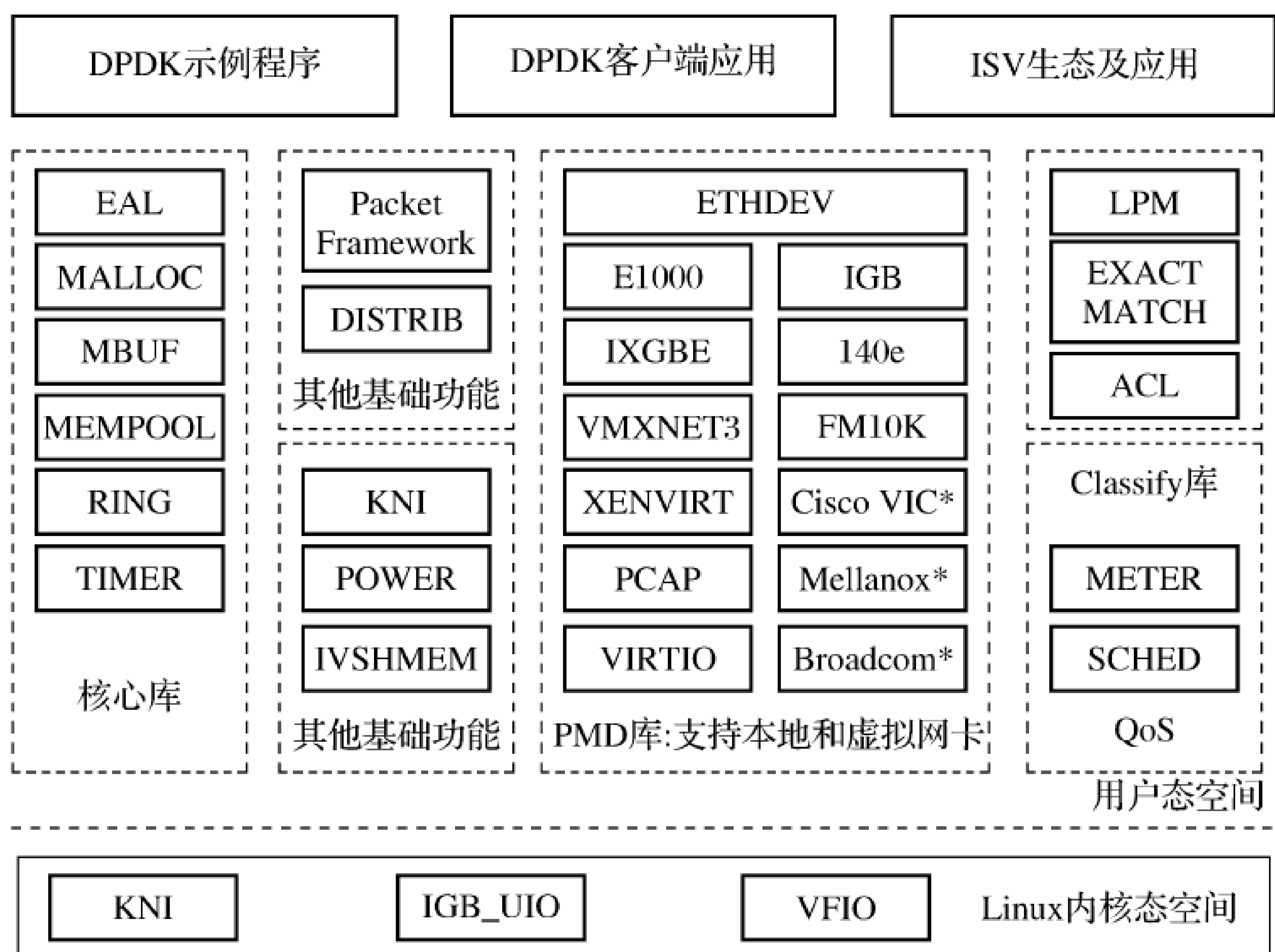


图 22－36　DPDK 框架视图

- **核心库(Core Libs)**：提供了系统环境抽象层(Environment Abstraction Layer, EAL)、大页内存机制、缓存池机制、定时器和无锁环形队列机制等基础加速功能。其中 EAL 也被称为环境适配层，它提供了一套 API，用于系统初始化、获取 CPU 核心数、线程数

等配置信息，并能发现外围的 PCI 设备、设置 Huge Page、预留与高速缓存相应的缓冲区和描述符环、初始化轮询模式驱动程序、生成工作线程等。EAL 在用户态空间的初始化过程如图 22－37 所示。

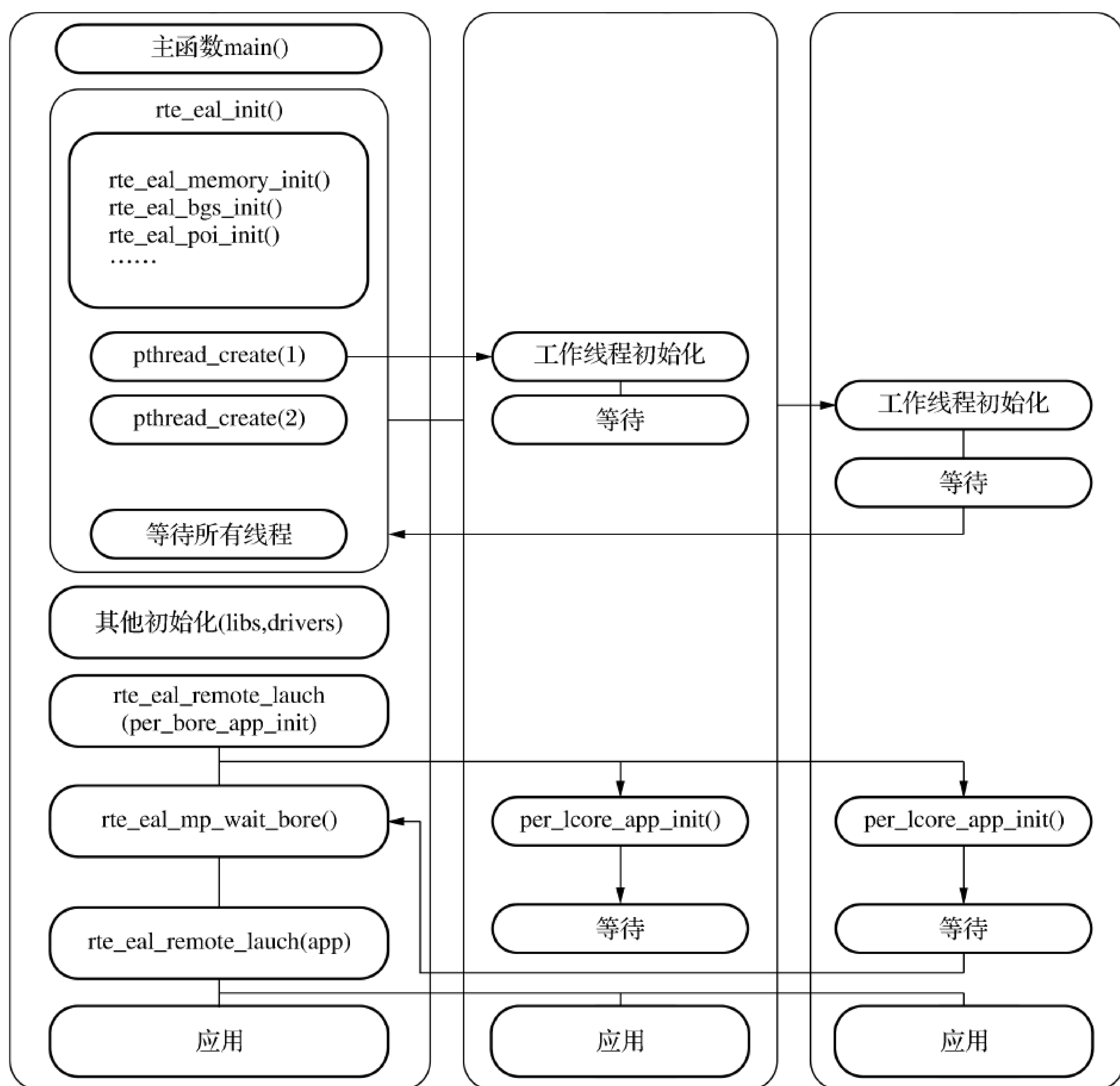


图 22－37　EAL 在用户态空间的初始化过程

- **PMD 库**：提供用户态轮询和处理器亲和性绑定功能，支持本地和虚拟网卡相关机制。
- **Classify 库**：支持精确匹配功能、最长匹配功能和通配符匹配功能，提供常用包处理的查表操作。
- **QoS 库**：提供网络服务质量相关的基础组件，包括限速机制等。
- **其他基础功能**：DPDK 也提供了若干平台特性，包括：
  - 与 Linux Kernel Stack 建立快速通道的 KNI（Kernel Network Interface，内核网络接口），通过 kni. ko 模块将数据报文从内核态协议栈传递到用户态，以便 socket 接口对报文进行处理。
  - 为构建更复杂的多核流水线处理模型而提供 Packet Framework 和 DISTRIB 等框架基础组件。例如 Packet Framework 可以将常见的网络行为拆分成不同的查表操作

和对应的动作,并将其聚合抽象为若干 pipeline 模块。

如图 22-38 所示是基于 RSS 和 Flow Director 两种多队列机制的 DPDK 框架。在一个 2 路 CPU 的平台中,每个 CPU 具有 4 个逻辑核,其中核心 0 总是用于 DPDK 框架的主线程(用于管理 I/O 工作线程),且 CPU 中的 4 个核心也都用于 I/O 工作线程,包括 0 号核心(主线程开销很小,因此 0 号核心也可以运行 I/O 工作线程以最大化地利用计算资源)。

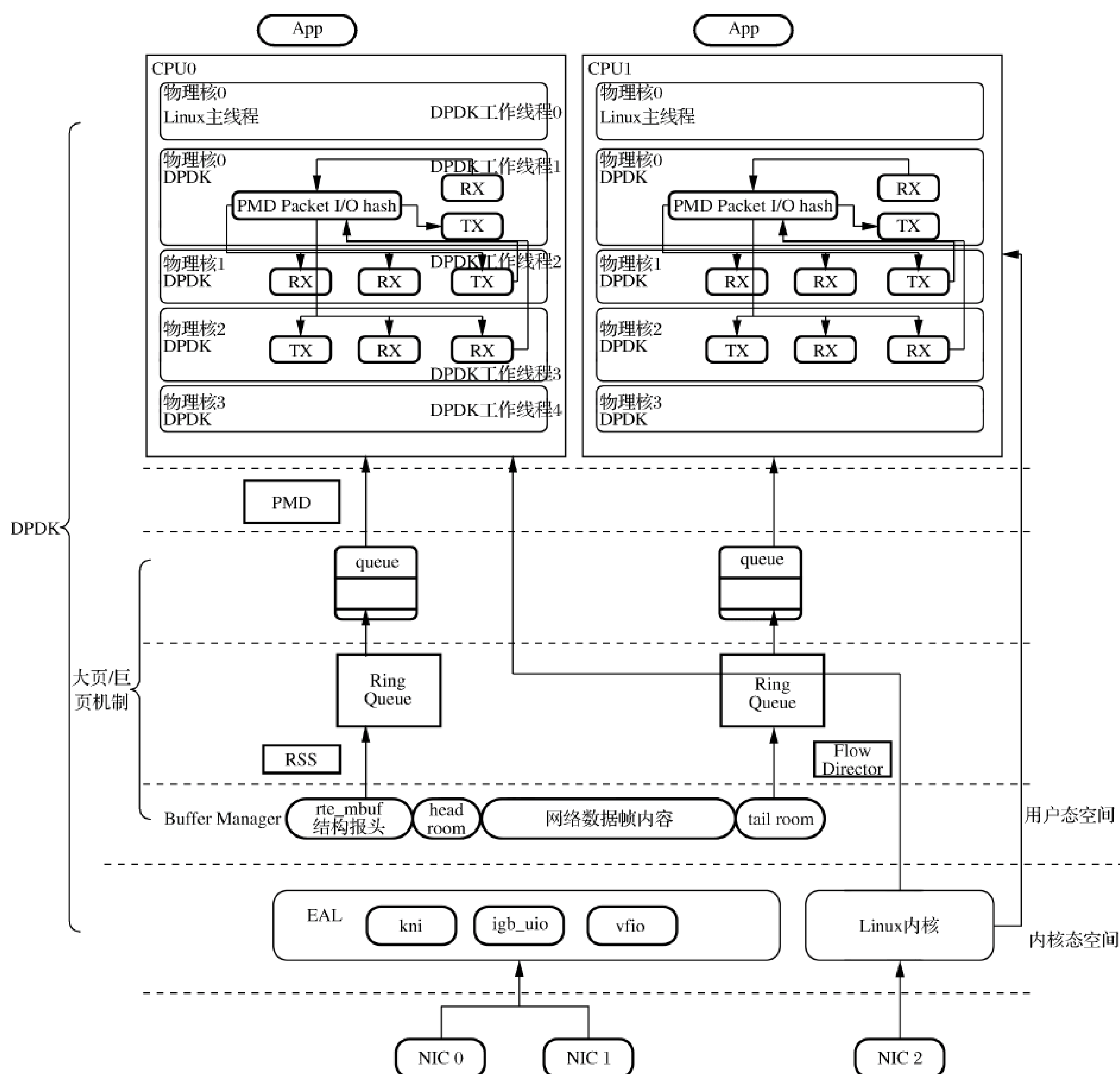


图 22-38　支持 RSS 和 Flow Director 两种机制的 DPDK 框架

22.2.1.4　基于 DPDK 框架的报文传输

DPDK 框架具有两种报文转发模型:运行至终结(Run To Completion,RTC)模型和流水线(Pipeline)模型。

RTC 模型针对的是通用型线程。在这种模型中,一个线程的处理步骤不管分为多少逻辑阶段,都会在一个 CPU 核上运行且运行到生命周期结束。RTC 模型中每个报文的生命周期只能在一个线程中出现,每个物理核都负责处理整个报文从 RX 到 TX 的生命周期。

Pipeline 模型将某个功能拆解成相互独立的多个阶段,阶段与阶段之间通过队列传递报

文，就像工厂流水线一样每个步骤都由专门的阶段负责完成，例如一个任务中的计算密集型和 I/O 密集型子任务就可以分为两个步骤/阶段并由不同的处理器执行。当然，能够这样拆分的前提条件是两个阶段可以独立拆解，阶段之间没有数据相关性。一般来说，对于网络数据包的处理，Pipeline 模型的效率要比 RTC 模型高很多。

DPDK 中的 Pipeline 模型也被称为 Packet Framework。在 DPDK 框架中，流水线上的每一个模块都是一个处理引擎，只处理特定的事务，每个引擎都有输入和输出，引擎之间就是通过这些输入和输出连接起来的。

采用 Pipeline 模型，可以将一些特定的步骤“卸载”给专用的芯片执行，例如 FPGA。通过专用硬件芯片实现通用“CPU + 软件”才能实现的某些步骤，其效率必然要高很多，让最专业的部件去做最擅长的事，也提高了计算资源的利用率。图 22 - 39 中将 RTC 模型与 Pipeline 模型进行了对比。

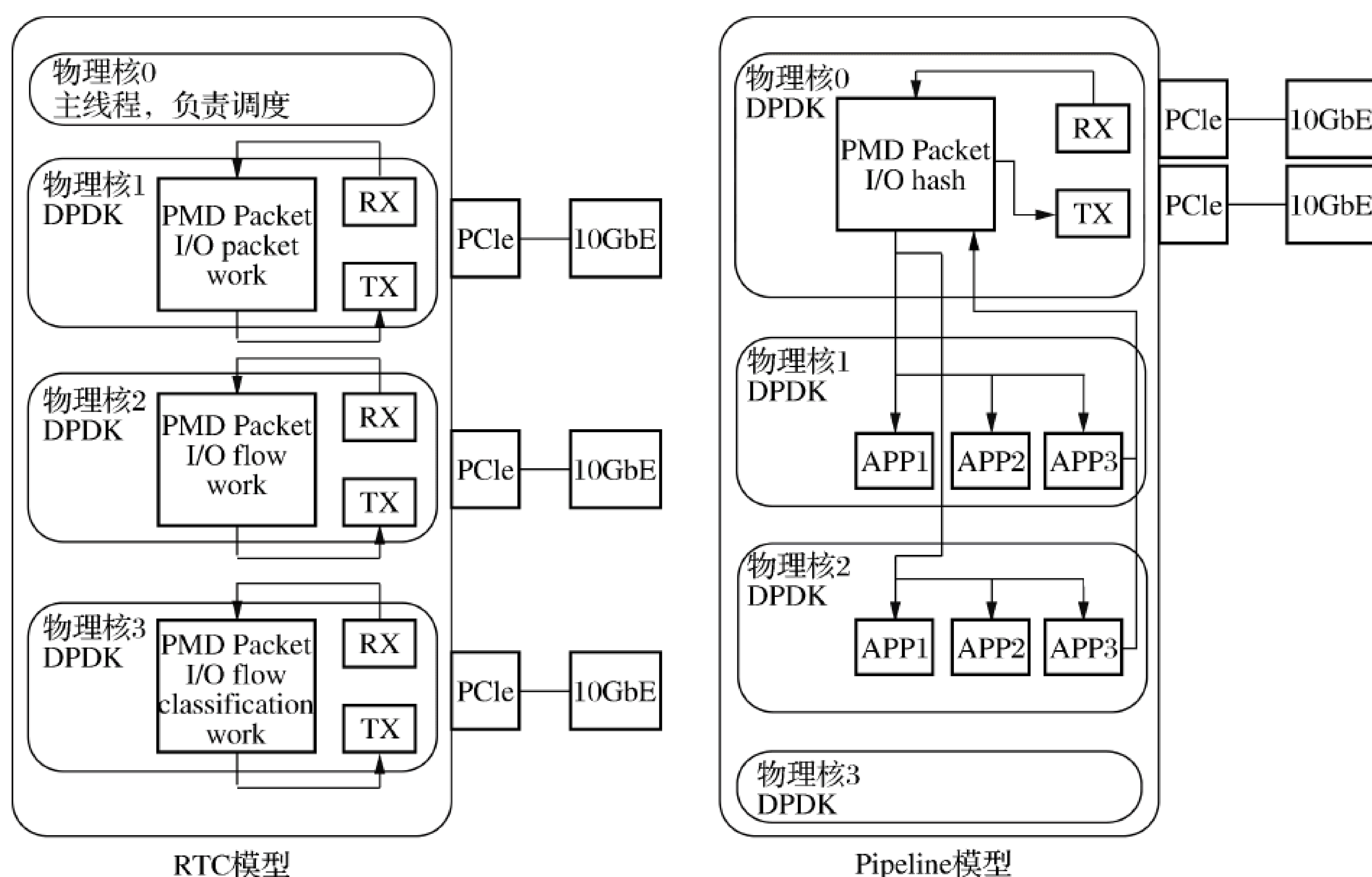


图 22 - 39　RTC 模型与 Pipeline 模型的对比

22.2.2　SPDK 框架

在云存储和存储加速技术高速发展的时代，提升存储效率和性能已成为数据平面 I/O 加速的必由之路。在存储软件栈体系中，限制存储效率和性能的因素包括磁盘读写速度、磁盘驱动运行效率、文件系统效率以及连接通道的传输效率。其中连接通道既包括 SCSI 接口控制器、光纤控制器等 DAS（Direct Access Storage，直接存取存储）环境下的传输通路，也包括以太网、光纤通道交换机等 NAS（Network Attached Storage，网络附加存储）环境下的传输通路。而提升存储性能既包括硬件层面的扩容提升，也包括软件层面的优化改进。

由于存储设备等硬件的性能飞速提升，相较而言存储软件栈的效能提升就不是那么尽如人意了，因此软件方面的提升就成了当务之急。在数据平面软件栈的优化和提升方面DPDK已经迈出了坚实的一步，为网络I/O加速做出了很好的榜样，而SPDK可以站在DPDK的肩膀上，利用DPDK的既有成果在存储数据平面I/O加速技术上开辟新的可能。

SPDK，即 Storage Performance Development Kit（存储性能开发工具包），是Intel提供的一套专门用于IA（Itanium Architecture，安腾架构）平台的高性能存储方案，也是针对基于NVMe SSD作为后端存储媒介的应用软件加速库。这里的高性能是指充分发挥IA平台的硬件加速特性（如Intel IOAT，英特尔I/O加速技术）和软件优化后的性能增量以提升存储的I/O效率。

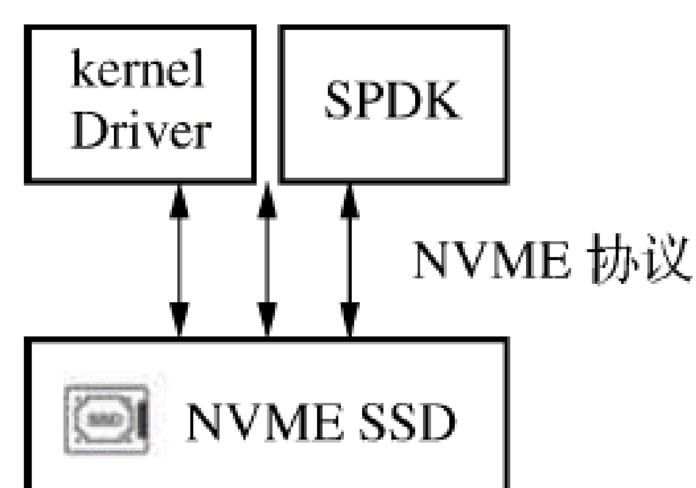


图22－40　SPDK与NVMe SSD的关系

SPDK的关键技术除了基于SSD和IOAT的一系列硬件加速特性外，软件层面则包括运行于用户态（UIO机制）、轮询机制（PMD机制）、缩短驱动软件栈深度（缩短I/O路径）、无锁化机制等，可见SPDK确实是基于DPDK一系列优化成果做出的改进。

在介绍SPDK框架之前，我们先来介绍一些存储领域的预备知识。

22.2.2.1　预备知识

1. RDMA技术

RDMA（Remote Direct Memory Access，远程直接内存存取）是DMA机制的远程实现版本。其作用是使计算机可以跨系统存取其他计算机的内存而无需消耗处理器计算资源和时间片，当然也无需操作系统的参与（这是因为操作系统参与会消耗处理器时间片和计算资源）。简单地说，RDMA技术是发生在跨系统网卡与内存之间的事情。

从图22－41可以看出，基于协议栈的数据存取（左上方）既要跨越用户态与内核态空间，也要经过网络协议栈驱动的层层处理，而RDMA技术（右上方）则明显具有以下优势：

- **Kernel-Bypass**：在RDMA中应用程序直接操作设备接口，不需要通过系统调用在用户态与内核态之间来回切换，也不经过内核态协议栈（内核旁路），因此也就省却了上下文切换的开销。
- **Zero-Copy**：由于网卡可以直接与应用程序的内存互相传输数据，因此不需要在网络协议栈各层之间拷贝数据（零拷贝），这也缩短了数据流的处理路径。
- **None-CPU**：数据传输过程中，除了最开始的传输协商流程需要CPU的参与外，数据的拷贝完全由网卡而不是由CPU搞定，数据包的收发也无需中断服务例程的响应。
- **基于消息的事务**：在RDMA中数据是被作为消息处理的，这与一般的基于流的方式有很大不同，这意味着应用程序不需要将数据流分隔成不同的消息和事务。
- **分散/收集条目支持**：RDMA支持读取多个内存缓冲区的数据并将其合并为一个消息，也支持获取一个消息后写入多个内存缓冲区。

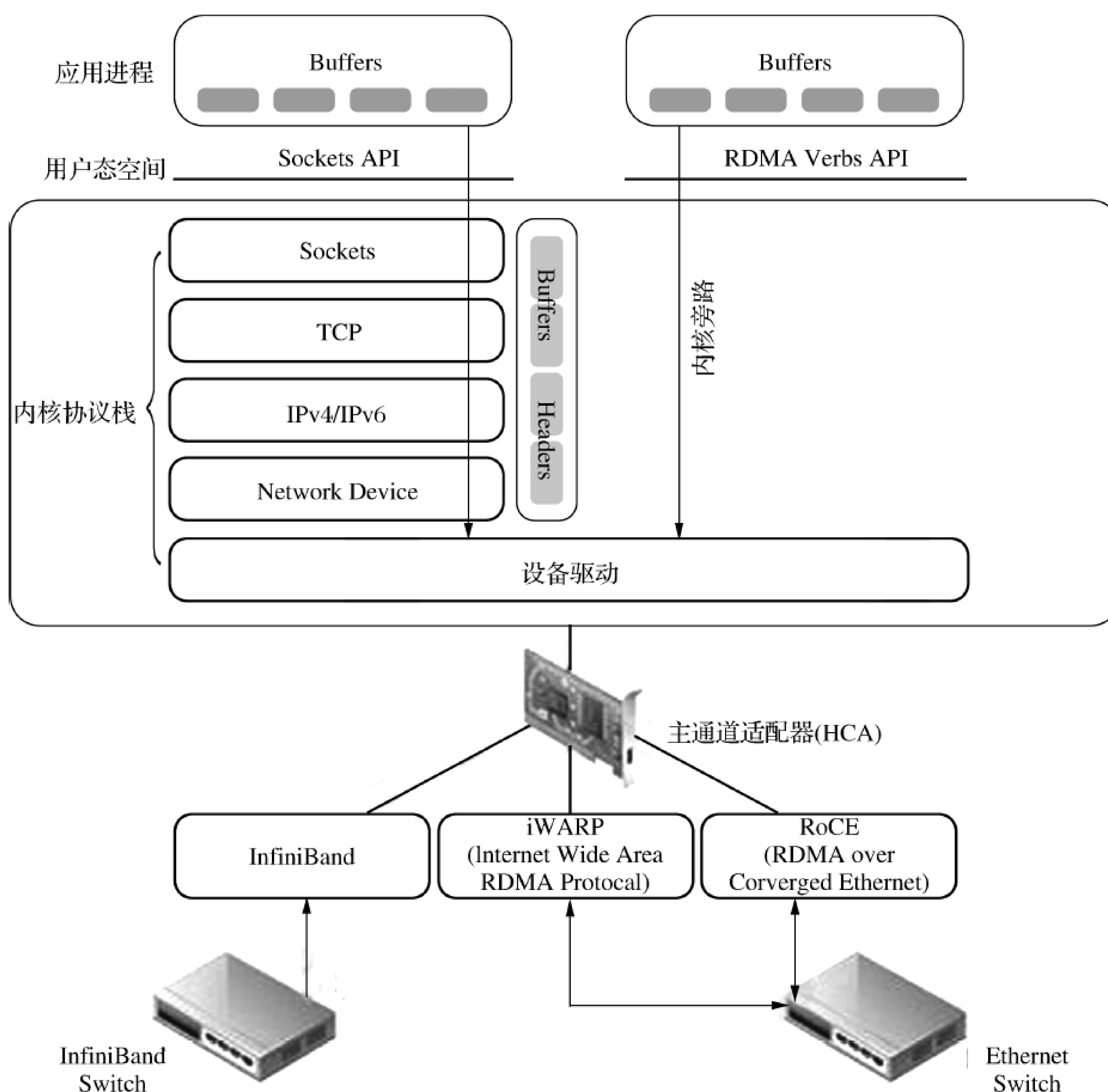


图 22－41　基于协议栈驱动和 RDMA 的远程数据存取

同时，应用进程只需要在本地网卡中注册对应的内存缓冲区就可以使用 RDMA 了，因此使用非常方便也是 RDMA 的一大优势。

RDMA 分为软件和硬件两部分，软件部分包括了 RDMA 服务向应用进程发布的 API 接口，这些接口被称为 Verbs(中文解释比较晦涩)。应用程序在使用 RDMA 时需要将 I/O 接口修改为 Verbs 接口，这是由于 RDMA 不再需要 I/O 管理器、文件系统、磁盘驱动等一系列操作系统软件栈了，因此与操作系统之间原有的接口必须改变。

RDMA 的硬件部分是指集成了 RDMA 功能的网卡适配器，这种适配器支持三种通信方式：InfiniBand、iWARP 和 RoCE，它们也是 RDMA 之下的协议，为 RDMA 提供了传输支持和链路实现。其中 InfiniBand 是 RDMA 原生支持的，iWARP 是基于 TCP/IP 协议的，RoCE 则是基于以太网的。表 22－4 中比较了这三种通信方式，图 22－42 则描述了支持这三种方式的 RDMA 架构。





表 22－4　RDMA 的三种通信方式的比较(表中内容来自 CSDN)

	InfiniBand(IB)	iWARP	RoCE
标准组织	IBTA	IETF	IBTA
性能	最好	稍差(受 TCP 影响)	与 IB 相当
成本	高	中	低
网卡厂商	Mellanox 40 Gbps	CHelsio 10 Gbps	Mellanoix-40 Gbps Emulex-10/40 Gbps

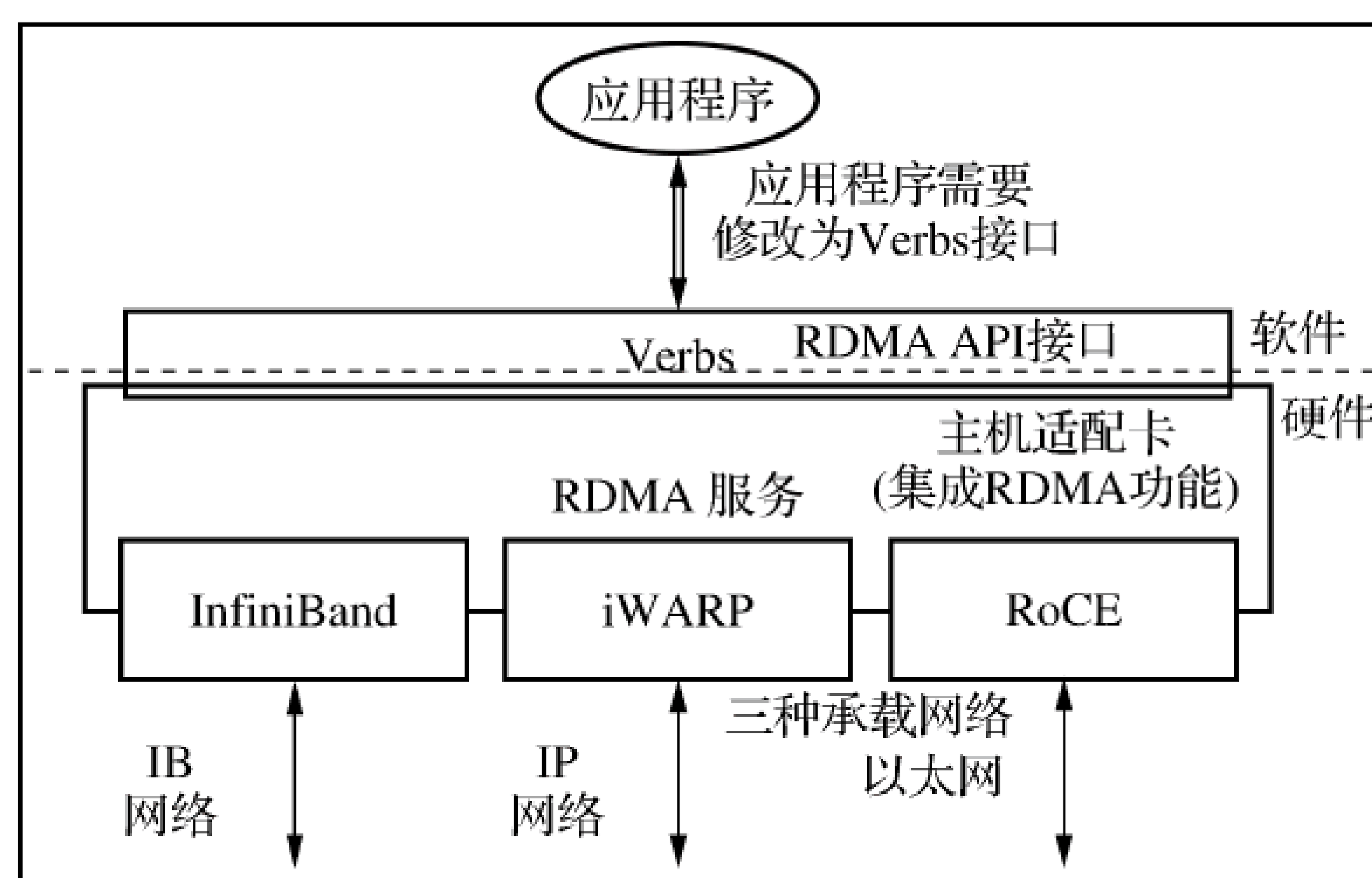


图 22－42　三种通信方式的 RDMA 架构

关于 RDMA 的软件部分,除了 Verbs 接口,其实还允许以另一种方式与应用进程交互:OFED 协议接口,支持 OFED 的 RDMA 架构如图 22－43 所示。

➢ 应用程序和具有 RDMA 引擎的以太网卡(RDMA-aware Network Interface Controller,RNIC)之间的传输接口层被定义为 Verbs。

➢ Verbs 接口是由 OFA(Open Fabric Alliance)发布的。Verbs 掩盖了底层硬件差异,无论是 InfiniBand 还是以太网都可作为 Verbs 的后端实现。

➢ RDMA 的基本操作接口分为两类:内存 Verbs(Memory Verbs)和消息 Verbs(Messaging Verbs),每一个 Verbs 可以理解为一个函数。

➢ RDMA 软件部分还包括一个 RDMA 协议栈,这个协议栈是由 OpenFabric 联盟(OFA)发布的,被称为 OFED(Open Fabric Enterprise Distribution)协议栈,并分为 Linux 和 Windows 两个版本,可以无缝兼容已有应用,并支持多种 RDMA 传输协议。

➢ OFED 是一组开源软件驱动、内核核心代码、中间件和支持 InfiniBand Fabric 的用户级接口程序。

➢ OFED 协议栈向上提供北向协议 ULP(Upper Layer Protocol,高层协议),上层应用程序不需要直接和 Verbs API 对接,而是借助于 ULP 直接与 RDMA 对接;向下则提供了 RNIC 实现 RDMA 和 LLP(Lower Layer Protocol,低层协议)等机制的消息队列服务。

➢ OFED 协议栈的北向协议具有更好的适配性和兼容性,一般的应用进程无需修改接口

就可以直接运行在 OFED 协议栈上。

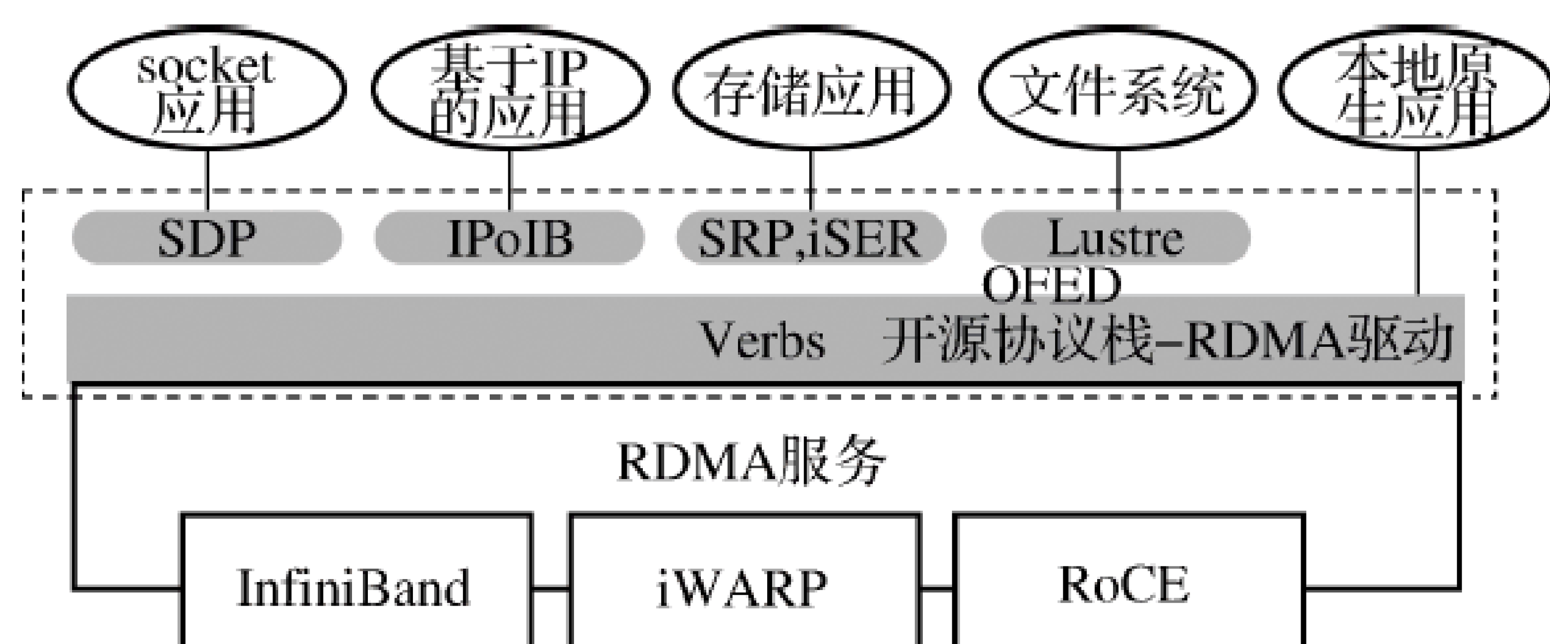


图 22－43　支持 OFED 开源协议栈的 RDMA 架构

OFED 可应用于 Linux 和 Windows，包括各种诊断和性能工具，这些工具可用于监视 InfiniBand 网络的运行情况，例如传输带宽和 Fabric 内部的拥塞情况。

OFED 可也是一个单一的软件栈，包括驱动、中间件、用户接口以及一系列的标准协议（例如 IPoIB、SDP、SRP、iSER、RDS、DAPL 等协议），并支持北向 MPI、Lustre/NFS over RDMA 等协议，也提供了 Verbs 编程接口。

RDMA 的通信是采用消息机制的，消息服务建立在通信双方本地端和远程端应用之间的 I/O Channel 连接之上，每次应用进程需要通信时就创建一条 Channel 连接通道，每条 Channel 的首尾两端各是一个队列对（Queue Pair，QP），每一端的 QP 由一个 SQ（Send Queue，发送队列）和一个 RQ（Receive Queue，接收队列）构成，这些队列中管理着各种类型的用于发送数据的消息。这些 QP 都被映射到应用进程的虚拟地址空间中，可以像访问虚拟地址一样访问这些 QP，也就相当于直接访问了 RNIC（RDMA 在 NIC 上的实现）网卡。RDMA 除了 SQ 和 RQ 外还有一个完成队列（Complete Queue，CQ），用于通知上层应用进程 QP 中的消息已经被处理完毕。一个 CQ 可以与多个 SQ 和 RQ 关联。

RDMA 中对数据的操作分为单边操作和双边操作两种方式。

- **单边操作**：只需要本地端明确信息的源地址和目的地址就可以完成收发操作，远端不需要感知本次通信。单边操作包括 READ 和 WRITE 两种，多用于 RDMA 数据报文，都是通过 RDMA 在 RNIC 与应用进程缓冲区之间完成的，远端 RNIC 会将完成事件封装为消息返回本地端。
- **双边操作**：需要本地端和远端的应用层参与才能完成收发操作，包括 SEND 和 RECEIVE 两种，多用于 RDMA 连接控制类报文。

应用进程使用 RDMA 是非常简单的，只需要注册内存（Memory Registration，MR）即可。但在传输过程中应用程序不能修改注册内存的属性，操作系统也不能对数据所在的内存执行页面倒换操作（非分页的常驻内存），也就是说必须保持虚拟地址和物理内存的映射关系不变。

2. InfiniBand

InfiniBand（简称 IB）是一种网络通信协议，提供了基于交换的架构，主要应用于存储领域的远程内存存取和计算机系统之间的互联，也是 RDMA 的支撑技术之一，其组网可参考

图 22－44。使用 IB 网络时需要原生硬件支持：需要配置支持 IB 协议的交换机（IBA 交换器），主机网卡也需要支持 IB 协议。

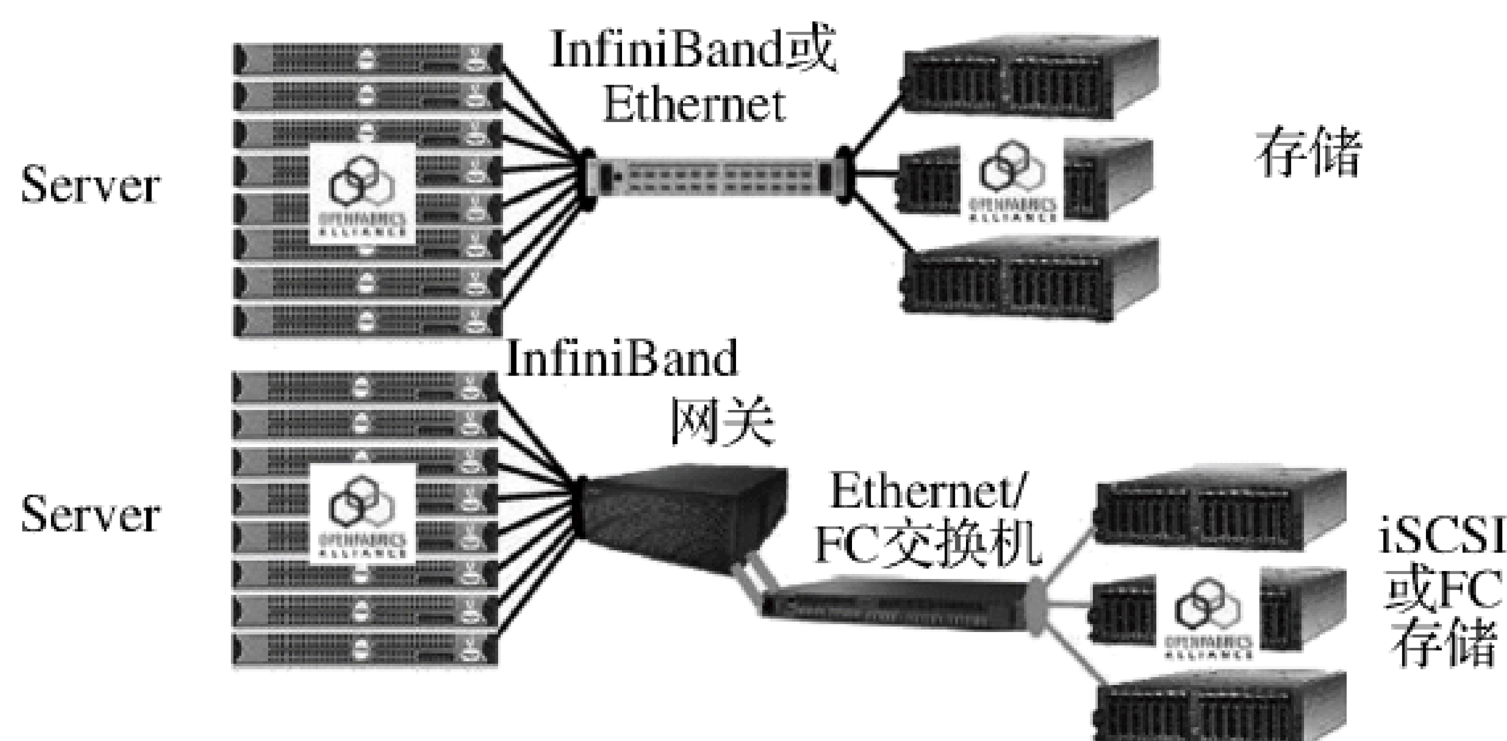


图 22－44　支持 IB 协议的存储系统组网（图片来自 CSDN）

InfiniBand 提供了以信用为基础的流控制通信手段。所谓以信用为基础，就是发送端节点不给接收端发送超出广播事先约定大小的数据包，这被称为“遵守信用”。既然这么守信，数据包自然也很少会丢失。同时，IB 也是以通道（Channel）为基础的双向串行式传输协议，通信时在交换节点之间创建一条私有的、受保护的通道。

IB 的通信具有以下特点：

- 基于 RDMA 机制的通信无需 CPU 参与，由 InfiniBand 适配器来管理和执行收发操作。
- InfiniBand 会确认接收端缓存是否具有足够空间，否则不会传送数据。
- 接收端在数据传送完毕后会返回通知报文以通知发送端，并标识发送端缓存空间的可用性。
- InfiniBand 适配器的一端通过 PCI-E 接口连接到 CPU 上，另一端则通过 InfiniBand 网口连接到 InfiniBand 网络上。
- InfiniBand 串行链路可以在不同的信令速率下运行，也支持捆绑在一起实现更高的吞吐量。

InfiniBand 协议之上可以承载更上层的应用协议，包括 SDP、SRP、iSER、RDS、IPoIB 和 uDAPL 等，这些应用协议同时也是 RDMA 之上的协议。

- **SDP**：Sockets Direct Protocol，支持用户既有的基于 TCP/IP 协议的程序运行在 InfiniBand 高速网络之上。
- **SRP**：SCSI RDMA Protocol，在 InfiniBand 协议中将 SCSI 命令封包，允许 SCSI 命令通过 RDMA 在不同的系统之间传输。
- **iSER**：iSCSI Extensions for RDMA，类似于 SRP 协议，是 IB SAN 的一种应用通信协议，将 iSCSI 协议的命令和数据通过 RDMA 方式运行于 Infiniband 网络上。目前 iSER 作为 iSCSI RDMA 的存储协议已被 IETF 标准化。
- **RDS**：Reliable Datagram Sockets，是 Oracle 发布的运行在 InfiniBand 之上的 IPC（进程间通信）的协议，与 UDP 类似，用于在 InfiniBand 上使用 socket 机制收发数据。
- **IPoIB**：IP-over-IB，为了实现 InfiniBand 网络与 TCP/IP 网络兼容而制定的协议。IPoIB





是基于 TCP/IP 的协议,对于应用进程是透明的,原有的使用 TCP/IP 协议栈的应用程序不需要任何修改就能使用 IPoIB。

- **uDAPL**:User Direct Access Programming Library,这是一种允许用户程序通过直接访问动态库来实现 RDMA 通信的方式。

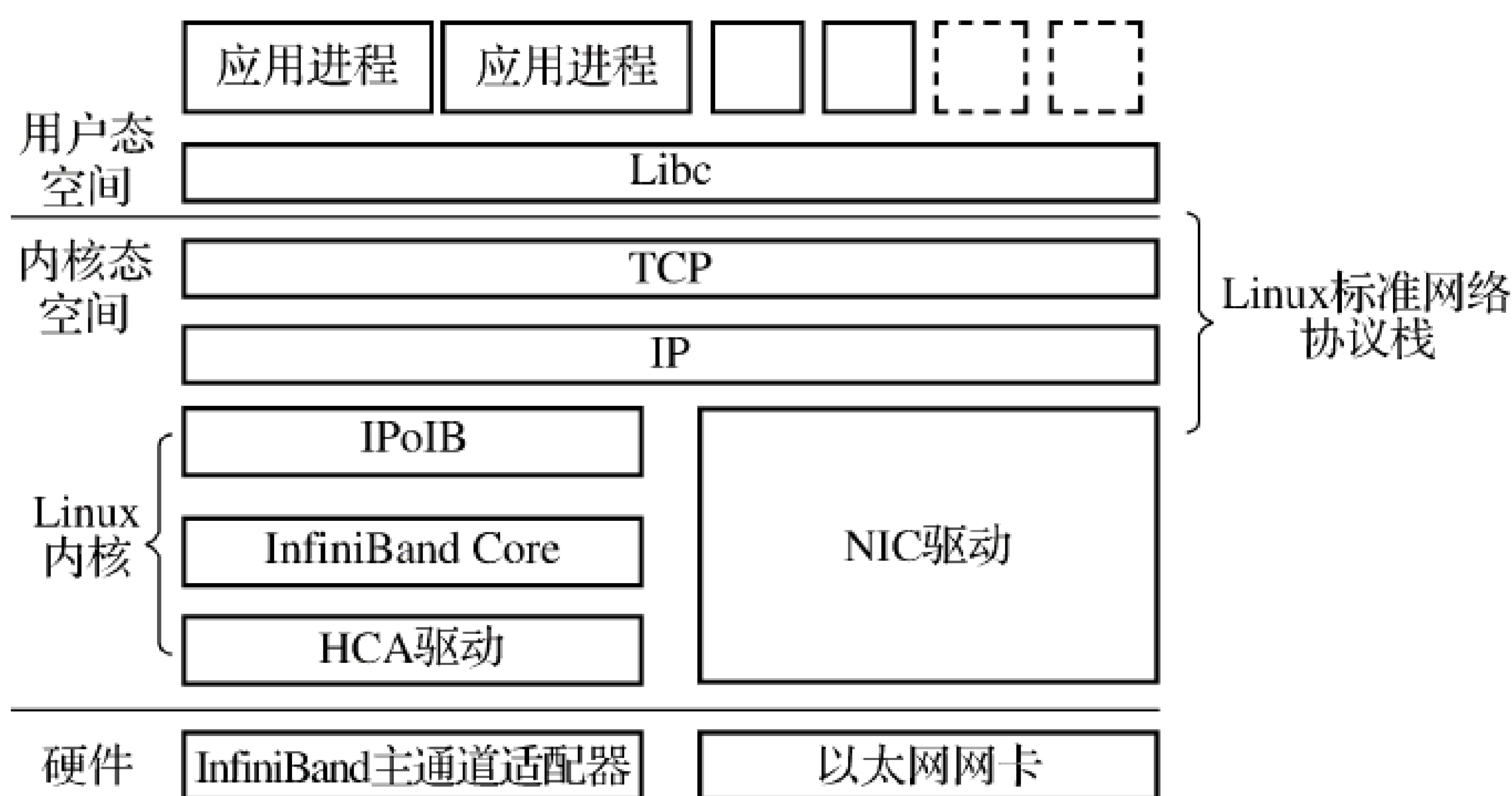


图 22－45　基于 InfiniBand 和 Ethernet 两种方式的软件栈

InfiniBand 软件栈可以分为用户态和内核态两部分,图 22－45 所示是基于两种方式的软件栈。用户态部分包括基于 TCP/IP 的应用程序、基于 SCSI 的应用程序、基于 iSCSI 的应用程序、基于 socket 的应用程序和基于文件系统的应用程序。内核态部分则包括 HCA(Host Channel Adapter)驱动程序、InfiniBand 核心模块和上层协议,其中 InfiniBand 核心模块最主要的部分就是 InfiniBand 设备的内核级中间层,该中间层允许访问多个 HCA NIC 并提供一组共享服务,这些服务包括:

- **通信管理(CM)**:提供了允许客户端建立连接所需的服务。
- **子网管理员客户端(SA Clinet)**:提供了允许客户端与子网管理员通信的功能。SA 包含建立连接所需的重要信息,如路径记录等。
- **子网管理器代理(SMA)**:用于处理子网管理数据包,允许子网管理器在每个主机上查询和配置设备。
- **性能管理代理(PMA)**:用于处理检索硬件性能计数器的管理数据包。
- **管理数据报服务(MAD Services)**:提供了一组允许客户端访问特殊的 InfiniBand 队列对(QP)的接口。
- **通用服务接口(GSI)**:允许客户端在特殊的 Admin QP(QP1)上收发管理数据包。
- **队列对重定向高层管理协议(QP Redirection)**:将对特殊 QP1 的访问重定向到专用 QP,这也是带宽密集型高级管理协议所需要的。
- **子网管理接口(SMI)**:允许客户端在特殊的 QP0 上收发数据包,该接口一般被子网管理器使用。
- **Verbs**:对中间层发布由 HCA 驱动程序提供的 Verbs 接口。InfiniBand 体系结构规范定义了 Verbs,Verbs 是必须提供的函数的语义描述,中间层将这些语义描述转换为一

组 Linux 内核 API。

InfiniBand 的软件架构如图 22－46 所示。

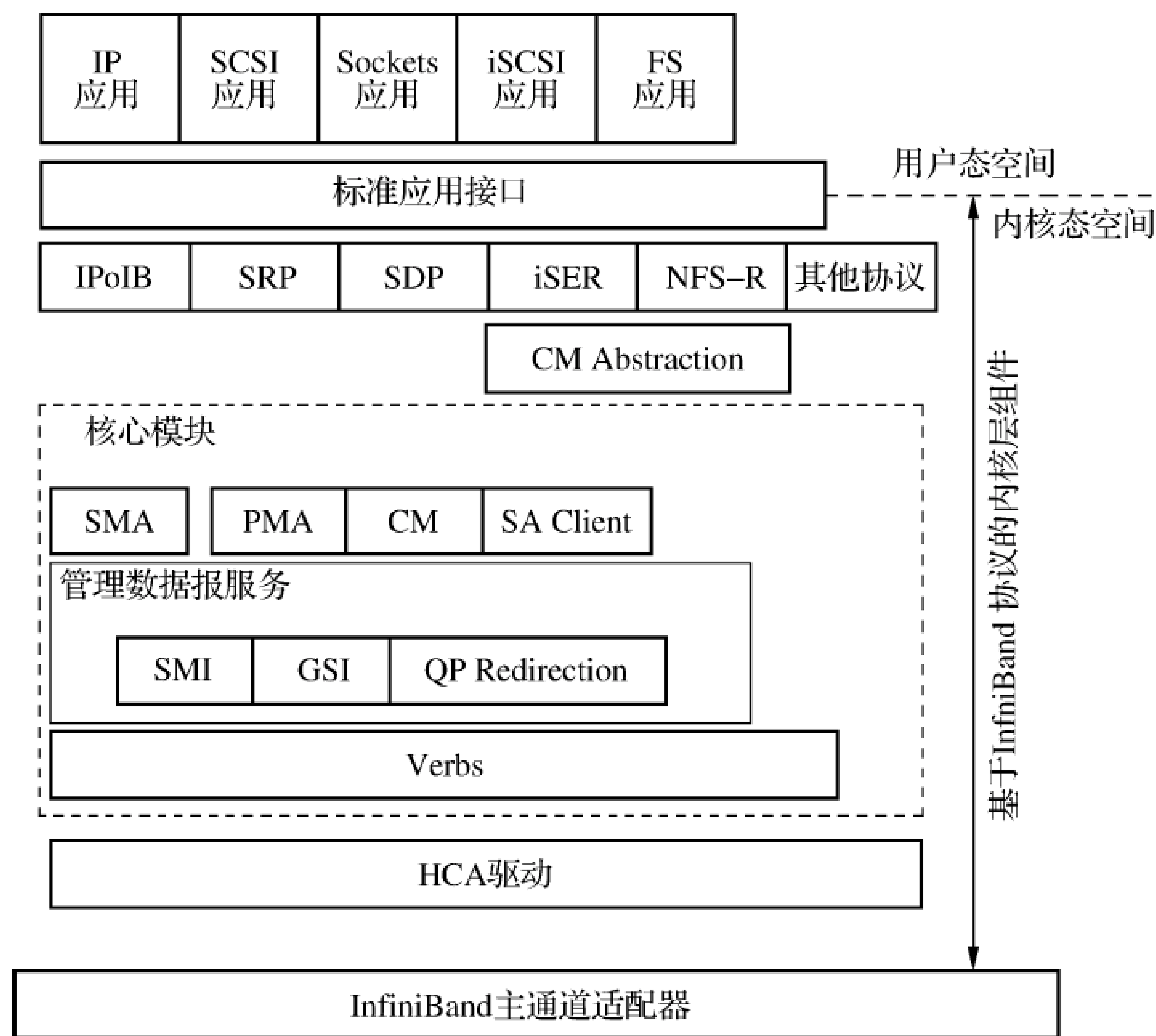


图 22－46　InfiniBand 软件架构

内核级中间层还负责在进程异常终止或客户端关闭后，对没有释放的已分配资源进行跟踪、引用计数和清理。

InfiniBand 堆栈的最低层由 HCA 驱动程序组成，每个 HCA 设备都需要一个特定于 HCA 的驱动程序，该驱动程序注册在中间层，并提供 InfiniBand Verbs。

3. RoCE

RoCE 即 RDMA 融合以太网（RDMA over Converged Ethernet），是一种基于以太网的 RDMA 通信协议，可在没有原生硬件支持的情况下使用。RoCE 支持在标准以太网上运行 RDMA 的各种操作，当然这要求网卡必须支持 RoCE 机制。与基于 TCP/IP 的传统通信方式对比，RDMA 技术旁路了内核态的网络协议栈，具有无可比拟的性能优势，并且 RoCE 可以使 RDMA 协议具有更强的普适性和通用性。基于 TCP/IP 和 RDMA 两种方式的远程通信如图 22－47 所示。

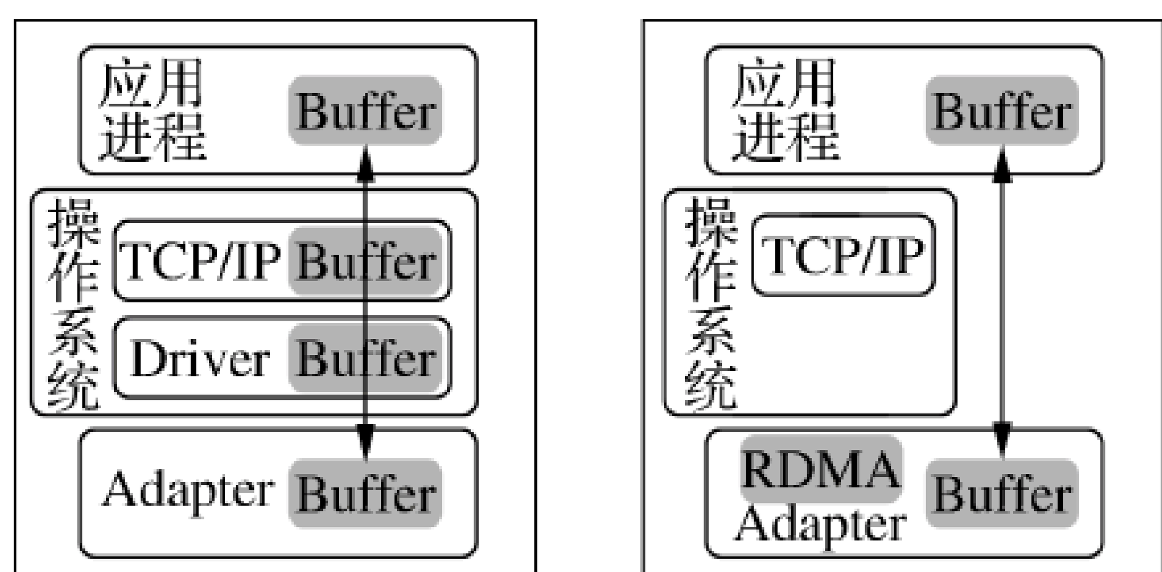


图 22－47　基于 TCP/IP 和 RDMA 两种方式的远程通信

为了实现 RDMA，RoCE 采用标准以太网头封装 IB 协议，因此外层网络头是以太网头，内层网络头是 InfiniBand 协议头，最里面则是 IB 协议的报文内容。这也意味着 RoCE 报文只能在二层网络上运行，无法通过三层设备的解析。

RoCEv2 是针对 RoCE 的改进版本，具有更好的适应性。RoCE 基于二层的 Ethernet 协议，而 RoCEv2 则是基于四层的 UDP 协议，因此可以跨二层组网，如图 22－48 所示。RoCEv2 与 RoCE 的协议栈对比如图 22－49 所示。



图 22－48　RoCEv2 协议的报文结构

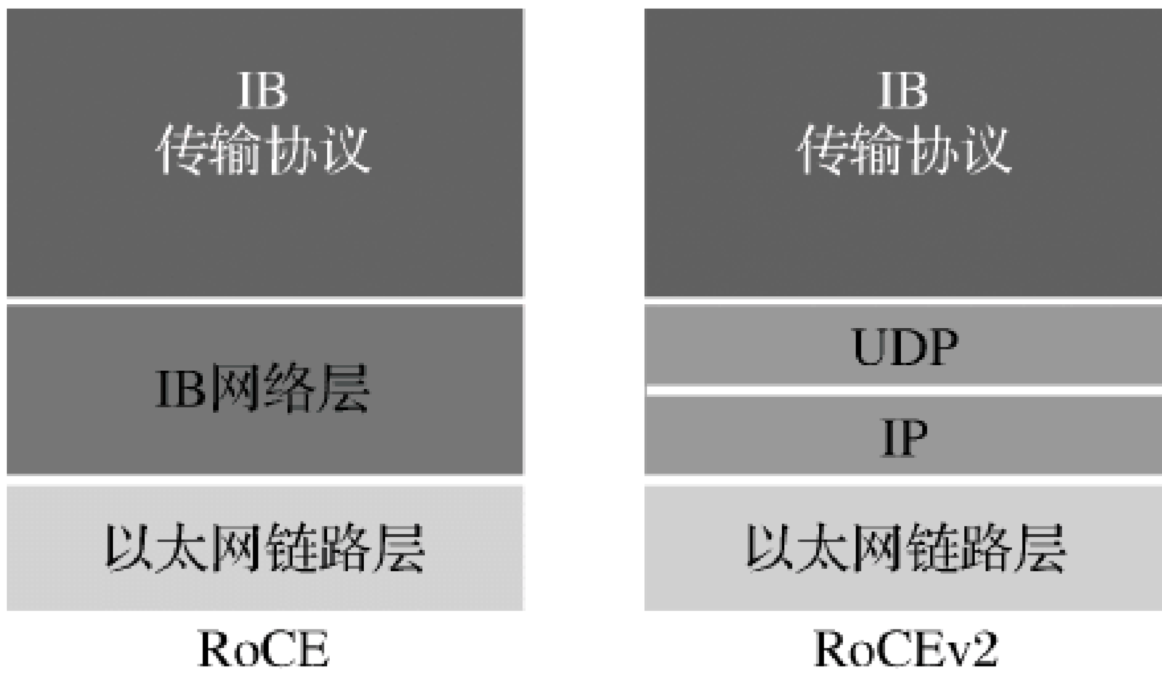


图 22－49　RoCE 与 RoCEv2 的协议栈对比

4. iWARP

iWARP 即互联网广域 RDMA 协议（Internet Wide Area RDMA Protocol），这是一种基于 TCP/SCTP 的 RDMA 通信协议，可在没有原生硬件支持的情况下使用。iWARP 可以基于 TCP/IP 协议执行 RDMA 的各种操作（RDMA over TCP/IP），由于跨越了二层协议，因此也可以在标准以太网基础上使用 RDMA 协议。不过在开启 CPU 卸载（例如 TCP 负载减轻引擎，TCP Offlod Engine，TOE）功能的前提下，iWARP 仍然要求网卡支持 iWARP（否则仍然需要软件实现 iWARP 的功能堆栈），因而失去了性能优势。软件方式与 TOE 方式的比较参见图 22－50。

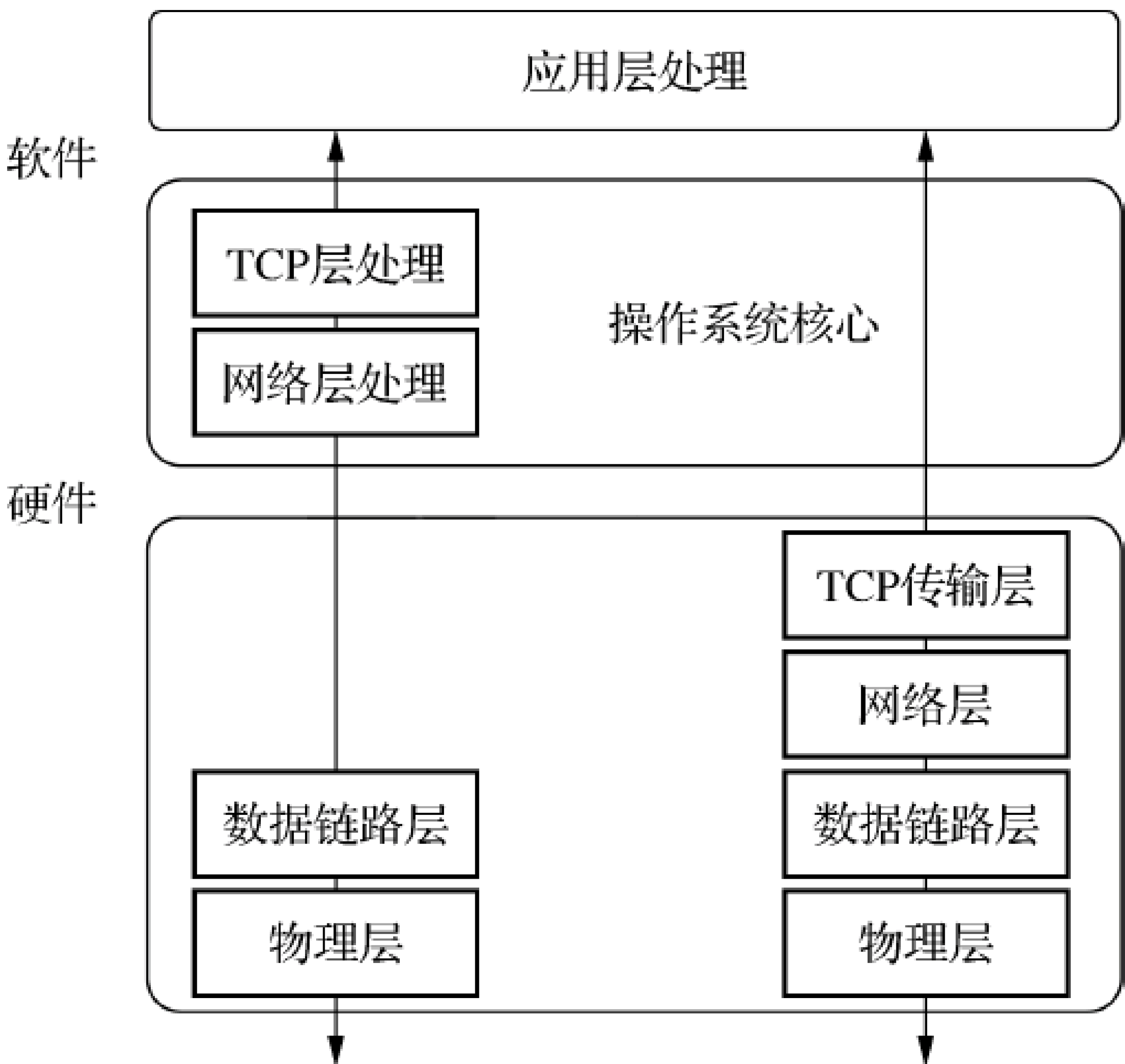


图 22－50　软件方式与 TOE 方式对比

5. 协议普适性

从 InfiniBand、RoCE、RoCEv2 和 iWARP 这 4 种协议栈的对比来看(如图 22-51 所示),后三者都是基于以太网协议的,并且 iWARP 和 RoCEv2 是基于三层的 IP 协议以及四层的传输层协议(TCP、SCTP、UDP)的,因此后两者具有更好的普适性。在四层协议的协议体内部,RoCE 和 RoCEv2 封装的是 IB 协议,iWARP 封装的则是 iWARP 协议。

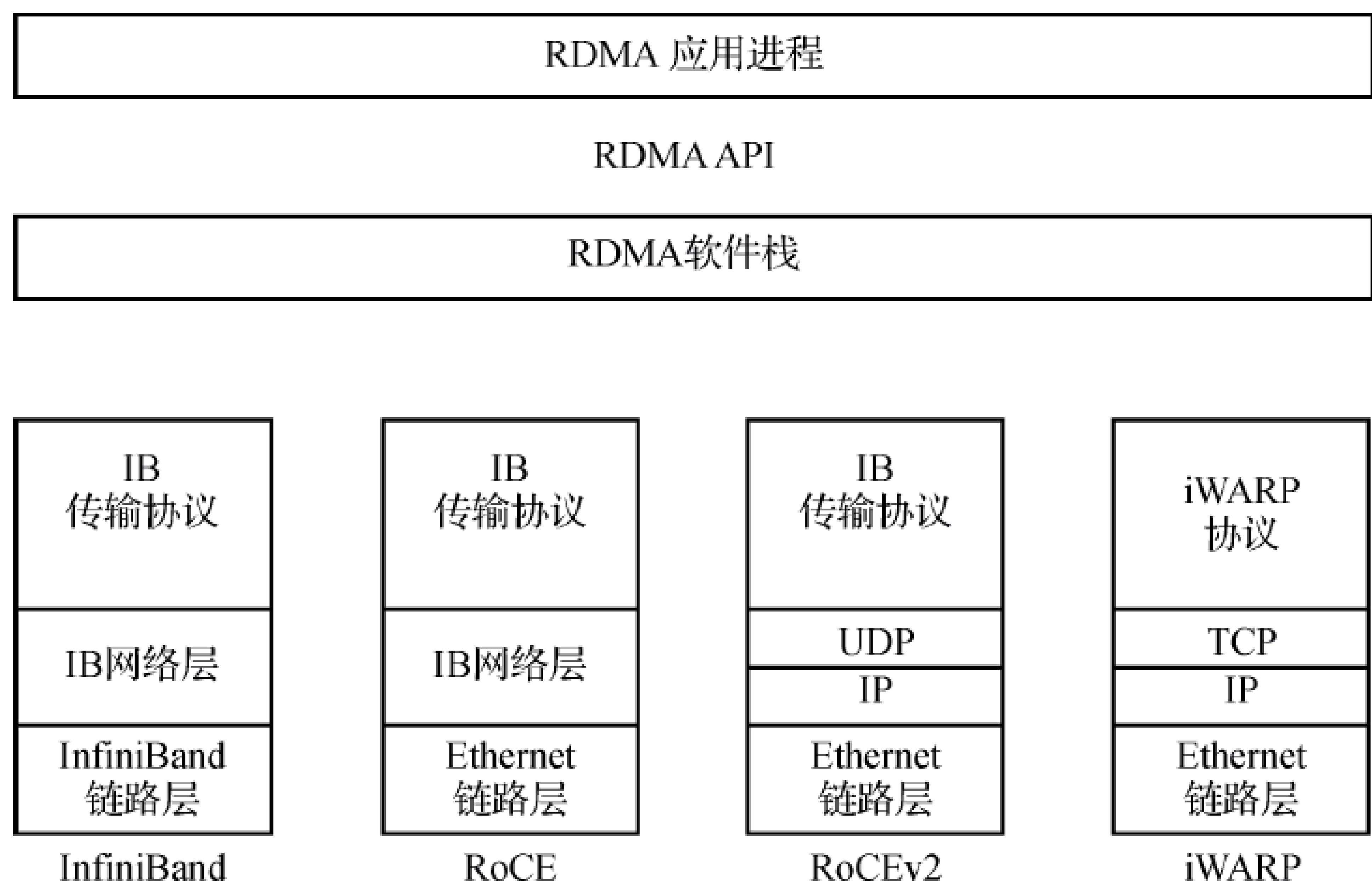


图 22-51 InfiniBand、RoCE、RoCEv2 与 iWARP 协议栈对比

6. HDD 与 SSD

HDD 即机械硬盘(Hard Disk Drive),读写时一次只能读取一块数据,这是因为读写时必须进行旋转以定位到第一个数据块的正确物理位置,然后再次旋转移动到第二个数据块的正确位置,以此类推。因此 HDD 对数据的读写效率不高。

SSD 即固态硬盘(Solid State Drive),是一种以半导体闪存(NAND Flash)作为介质的存储设备。闪存介质中保存数据的基本单元称为 Cell,每个 Cell 通过注入和释放电子来记录数据。由于 SSD 以半导体存储数据,是纯电子电路实现的,并不需要移动磁头,因此可以同时从许多不同的位置读取数据,具有很高的读写性能。同时,SSD 控制器通过运行复杂的 FTL(Flash Translation Layer,闪存转换层)处理逻辑将逻辑块读写映射成 NAND Flash 读写。SSD 的硬件结构如图 22-52 所示。

不过 NAND Flash 天生就有电荷逃逸的问题,这个问题会导致存储介质上的数据发生变化,这叫作**比特翻转**,因此 NAND Flash 本身是不可靠的。为了在不可靠的介质中建立可靠存储,SSD 采用了 ECC(Error Checking and Correction,差错检测和修正)硬件单元来解决比特翻转问题,即每次存储数据的时候,ECC 硬件单元会为存储的数据生成 ECC 校验码,每当读取数据的时候,硬件单元根据校验码恢复被破坏的比特数据。ECC 硬件单元都是集成在 SSD 控制器里面的,是控制器的一部分。不过,当比特错误率达到一定程度之后,ECC 机制会失效。

表 22-5 中列出了 SSD 与 HDD 的性能对比。

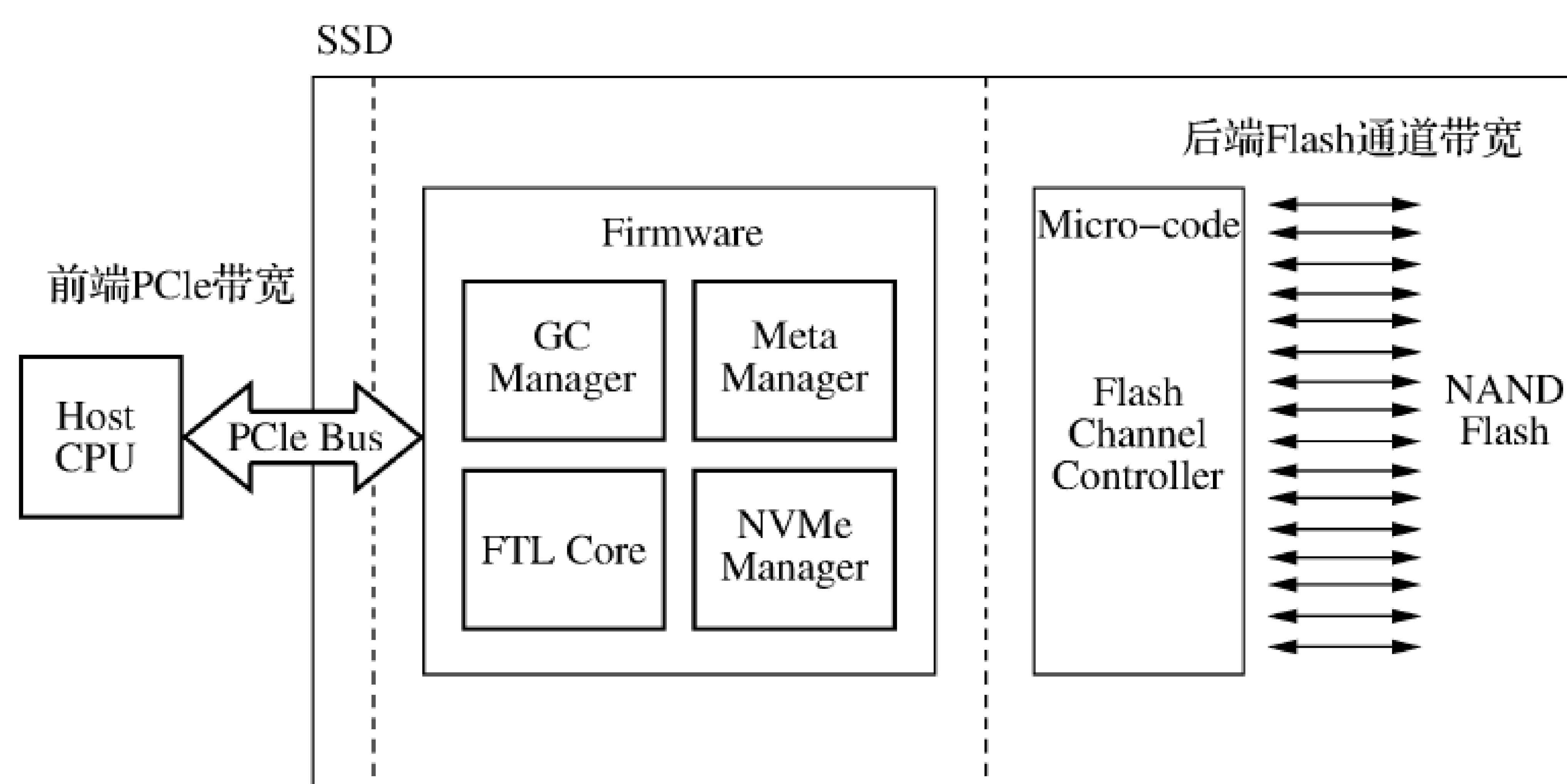


图 22－52　SSD 的硬件结构

表 22－5　SSD 与 HDD 的性能对比

	SSD	HDD(15K RPM)
IOPS	>10 000	220
时延	0.2 ms	5 ms

作为 SSD 存储介质的标配，3D NAND Flash 主要通过以下两种方式增加存储密度：

- 通过 3D 堆叠的方式增加 NAND Flash 的存储密度。
- 通过增加单 Cell 比特数来提升 NAND Flash 的存储密度。

2D NAND Flash 与 3D NAND Flash 的对比如图 22－53 所示。

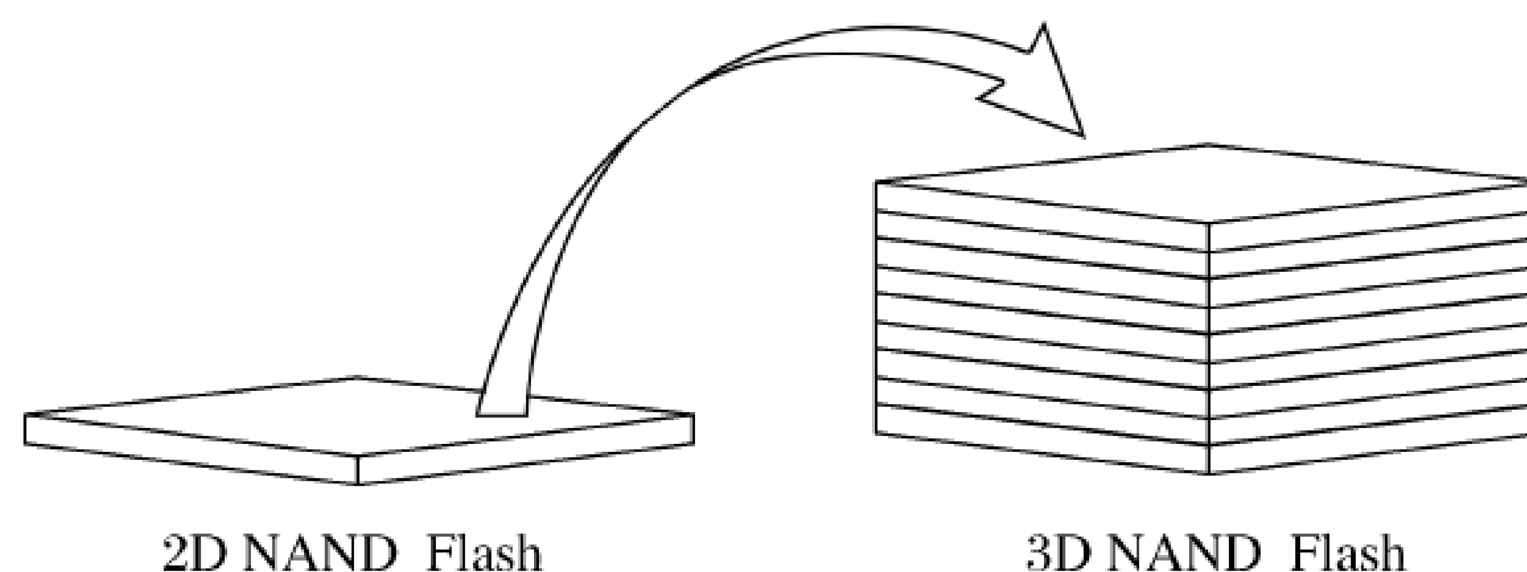


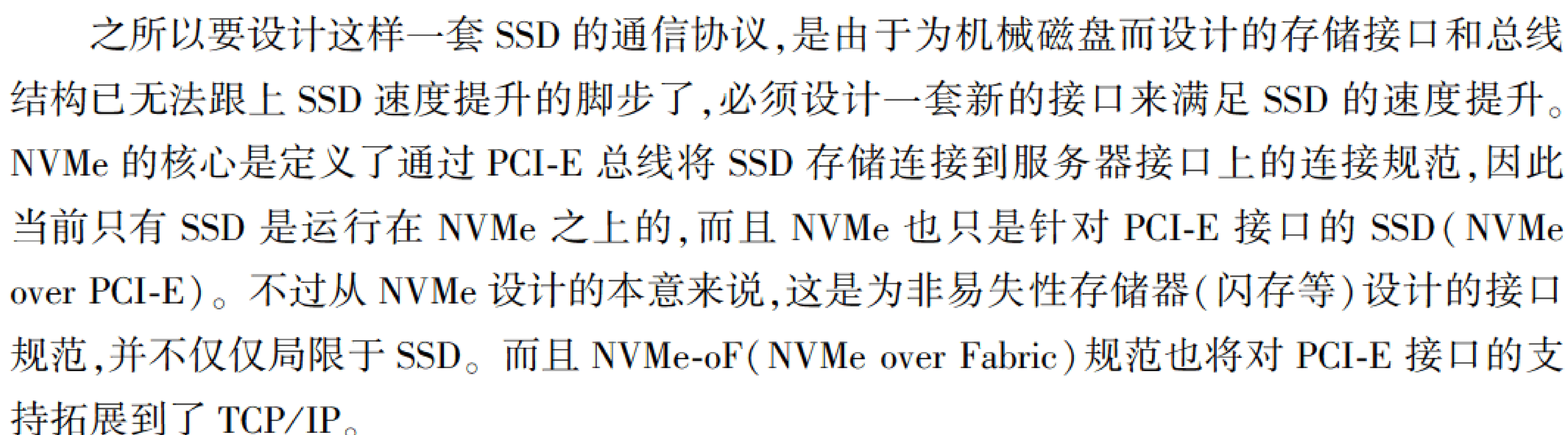
图 22－53　2D NAND Flash 与 3D NAND Flash 的对比

当前，一些新的非易失性内存技术也开始出现，例如 Intel 已经推出了 AEP 内存存储介质。可以预计，未来将会是非易失性内存和闪存共存的半导体存储时代。

7. NVMe

NVMe 即非易失性内存主机控制器接口规范（Non-Volatile Memory Express），这是一种专门用于加速 SSD 运行的协议，面向的是新型存储介质。NVMe 规定了操作系统与 NVM（非易失性存储）子系统之间的通信接口，也定义了一套与 AHCI（Advanced Host Controller Interface，高级主机控制器接口）规范不同的指令集和功能集。

使用 NVMe 与存储系统中的 SSD 通信，可提高每个处理器乃至整个存储系统的效率与性能，将存储设备拉近到 CPU 局部总线的距离，因此具有很高的 I/O 效率。



之所以要设计这样一套 SSD 的通信协议，是由于为机械磁盘而设计的存储接口和总线结构已无法跟上 SSD 速度提升的脚步了，必须设计一套新的接口来满足 SSD 的速度提升。NVMe 的核心是定义了通过 PCI-E 总线将 SSD 存储连接到服务器接口上的连接规范，因此当前只有 SSD 是运行在 NVMe 之上的，而且 NVMe 也只是针对 PCI-E 接口的 SSD（NVMe over PCI-E）。不过从 NVMe 设计的本意来说，这是为非易失性存储器（闪存等）设计的接口规范，并不仅仅局限于 SSD。而且 NVMe-oF（NVMe over Fabric）规范也将对 PCI-E 接口的支持拓展到了 TCP/IP。

NVMe 控制器通过两种方式提高 SSD 的 I/O 性能：

➢ 使用 PCI-E 总线连接 SSD 和系统 CPU，这种方式省略了 SATA 方式的一些步骤。不过 SATA SSD 已经可以满足大多数用户的需要了，并且比 NVMe SSD 便宜很多。
➢ 大规模多队列地实现 I/O 并行性。
  - 传统的 SATA 连接方式只支持一个队列，每个队列一次只能接收 32 条数据（队列深度为 32）。
  - NVMe 连接方式最多支持 64 K 个队列，每个队列一次可以接收 64 K 条数据（队列深度为 64 K，Admin 队列深度则为 4 K），因此理论上最大可支持 64 K × 64 K 条数据的并发。

NVMe 有两种命令，一种被称为 Admin Command，用于存储主机管理和 SSD 控制；另一种就是 I/O Command，用于主机和 SSD 之间数据的传输。同时，NVMe 还有三种命令队列：SQ、CQ 和 DB，它们在存储系统中的位置如图 22－54 所示。

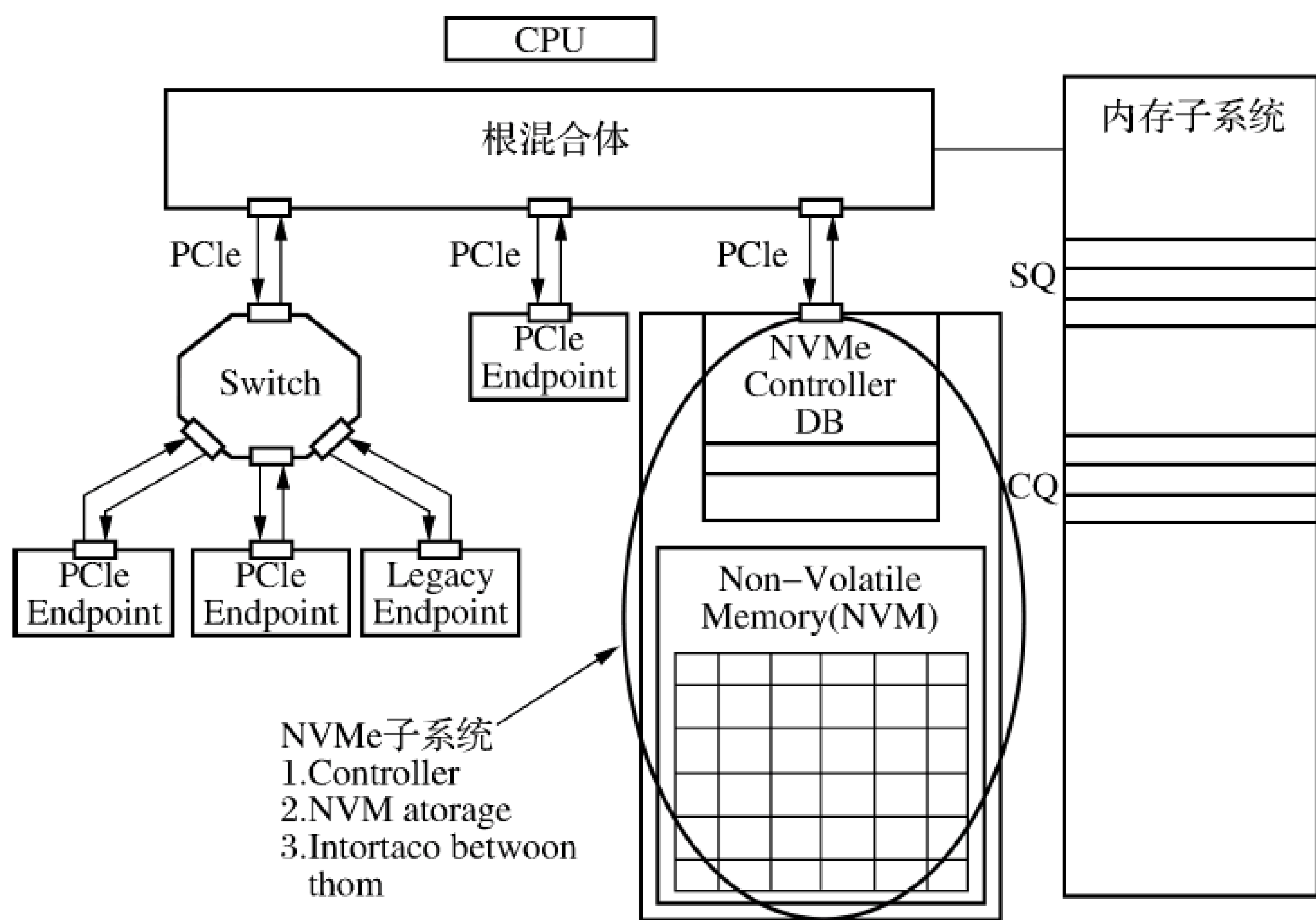


图 22－54　SQ、CQ、DB 在存储系统中的位置

➢ **SQ**：Submission Queue（提交队列），位于存储主机的内存中，主机要发送命令时先将命令存放在 SQ 中，之后通知 SSD 来取。
➢ **CQ**：Completion Queue（完成队列），位于存储主机的内存中，命令执行完成后，SSD 将

执行结果和完成状态写入 CQ 中,并通知主机到 CQ 中获取完成结果。CQ 与 SQ 是成对出现的,可以一对一,也可以一对多。

➤ **DB**:Doorbell Register(门铃寄存器),位于 SSD 控制器内部,是专用寄存器,用于存储主机通知 SSD 到 SQ 中取指令。

从上述分类可以看出,根据命令的不同队列可分为 Admin SQ、Admin CQ、I/O SQ 和 I/O CQ 4 种:Admin SQ 和对应的 Admin CQ 用来管理和控制 NVMe 控制器,例如创建和删除 I/O 队列,终止命令等;I/O SQ 和对应的 I/O CQ 则用来处理 I/O 命令。

系统在创建 SQ 前必须先创建对应的 CQ,执行删除操作时也要先删除 SQ 再删除 CQ。为了最大限度地减少加锁开销和多核间数据缓存的切换,一般会将一对队列绑定到一个处理器核上。也就是说,Admin SQ 和 Admin CQ 是一对一的,它们绑定在 0 号核上;I/O SQ 和 I/O CQ 可以一对一也可以多对一,绑定在其他几个核上,如图 22 - 55 所示。

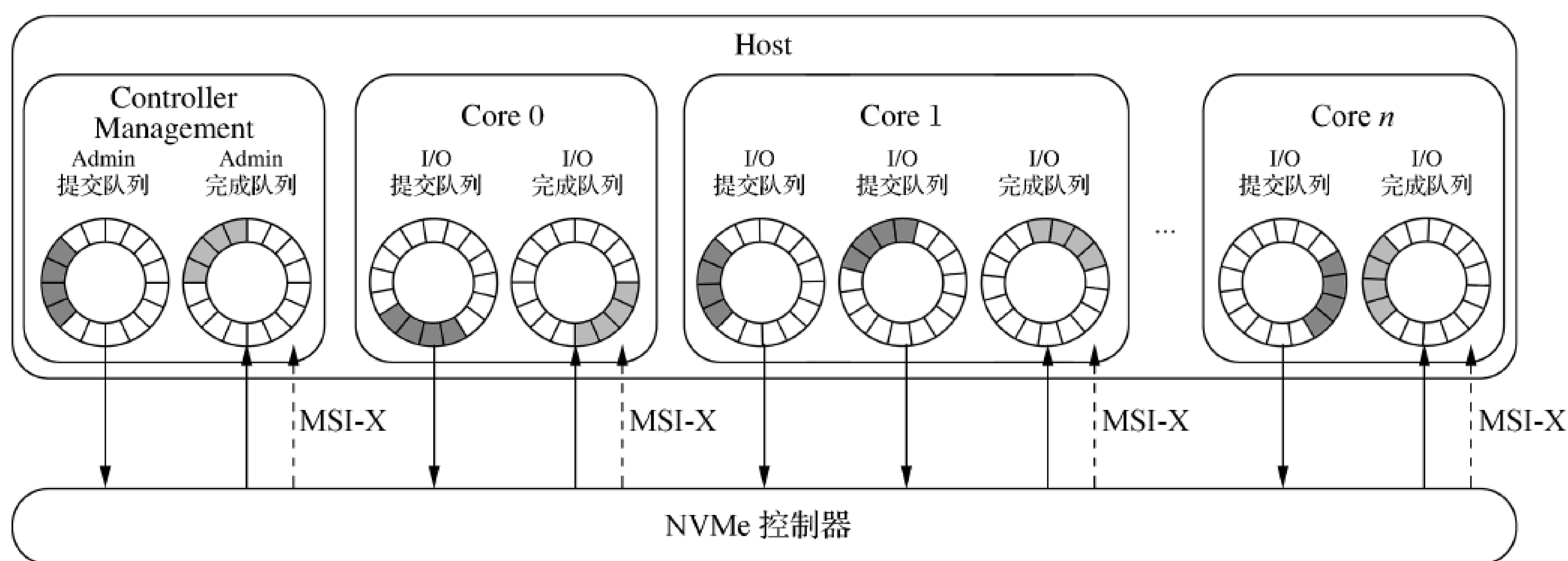


图 22 - 55　存储主机、NVMe 控制器与队列之间的绑定关系

除了上述提到的基于成对的 SQ 和 CQ 的机制之外,NVMe 还具有以下特性:

➤ 支持多个命名空间。一定量的 NVMe 的集合可被格式化为多个逻辑块,这些逻辑块称为命名空间。一个 NVMe 控制器可以支持多个有不同 ID 的命名空间。

➤ 支持命名空间共享。多个主机可以通过不同的 NVMe 控制器接入同一个命名空间。

➤ 支持多路径 I/O,即一个主机和一个命名空间之间存在多条完全独立的 PCI-E 路径。

➤ 与传统 SCSI 体系相比较,主机一侧的 NVMe 子系统减少了 I/O 调度层和命令层,使得 I/O 路径更短。

➤ 由于 NVMe 重新设计定义了 I/O 队列及相应的仲裁机制,相较于传统的 SCSI 体系软件栈减少了实现排队功能的通用 I/O 的调度层。

NVMe 在数据传输过程中的位置如图 22 - 56 所示。

相较于 NVMe,原有的 AHCI 规范只定义了一个交互队列,那么主机与 HDD 之间的数据交互只能通过这一个队列通信,即使是多核处理器场景下也只能通过这一个队列。在磁盘存储时代,由于磁盘是慢速设备,所以一个队列也就够用了。但在 SSD 场景下,其半导体存储介质具有很高的性能,传统的 AHCI 规范的单队列模式已不再适用。

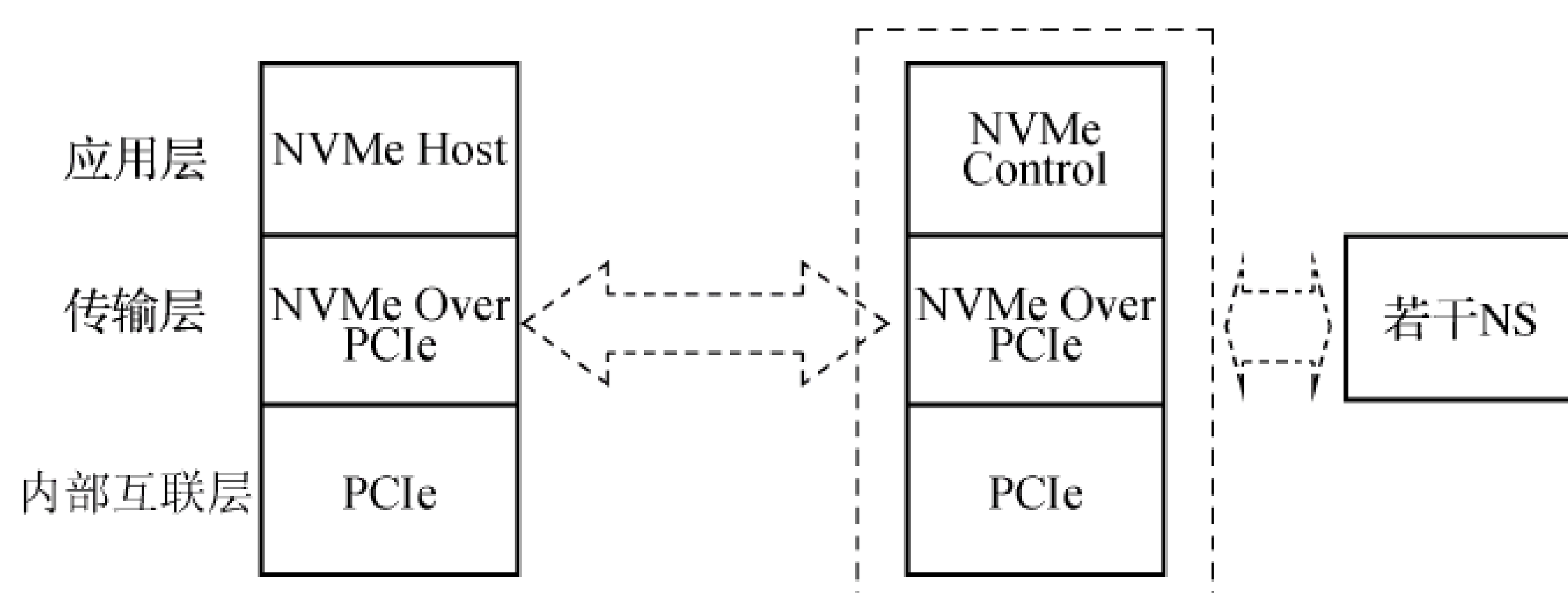


图 22－56　NVMe 在数据传输过程中的位置

22.2.2.2　SPDK 框架结构

1. SPDK 的设计特点

SPDK 本质上是利用了软硬件加速特性实现的存储框架，例如 TOE 和 DPDK 框架的某些模块等，是一种用于以 NVMe SSD 为后端存储媒介的应用软件加速库，因此也可以将 SPDK 看作 NVMe 驱动。SPDK 在以下方面具有高性能特点：

1）线程与处理器核的管理

- 基于 UIO 机制，不需要进行用户态到内核态的切换，从而避免了状态转换的开销。
- 依然采用轮询机制处理 I/O 操作（但工作线程会在 I/O 工作不饱和的情况下占满 CPU 使用率），相比于传统的基于中断框架的方式，这种方式在处理大并发量 I/O 时效率高得多。
- SPDK 依然采用了处理器亲和性机制，将工作线程与 CPU 核绑定。线程与核绑定后不需要对资源进行加锁，提高了线程的并行性。
- 在 SPDK 框架中提供了两种轮询器（Poller）：基于定时器的 Poller（框架中采用 spdk_poller 结构体来表示）和基于非定时器的 Poller，分别用于封装定时器事件处理例程和一般性事件处理例程。
- Poller 将 SPDK 用户指定的 I/O 处理函数、事件回调函数等封装起来形成一个结构体。工作线程（在 SPDK 框架中工作线程被称为 Reactor Thread）执行时会不停地轮询这些 Poller 并执行其中的函数。这有点像通信框架中的反应堆模型，只不过反应堆模型是在收到中断时被触发调用的，而 Poller 则是在主动轮询的条件下被一次次调用的。
- 每个 Reactor Thread 中有两个链表分别维护上述两种 Poller，每个链表可以保存多个同种类 Poller，同时 SPDK 也提供了 Poller 的注册和注销函数。

2）线程间通信

- SPDK 没有采用传统的基于锁、信号量等机制的线程间同步方式，而是采用了基于事件（Event）调用的同步通信机制。
- 在 Reactor Thread 对应的数据结构中（spdk_reactor 结构体）维护了一个 Event（spdk_event 结构体）环。这里的环是一种多生产者–单消费者模型的队列结构。

➢ 每个 Reactor Thread 都可以接受包括自身在内的任何 Reactor Thread 发来的消息。
➢ Event 结构包含了本 Reactor Thread 需要执行的函数、函数的参数以及该函数执行所要求的处理器核的信息。从本质上讲，线程间通信就是一种函数调用机制，caller 调用 callee 就是一种最基本的通信。

SPDK 框架中的线程模型如图 22 - 57 所示。

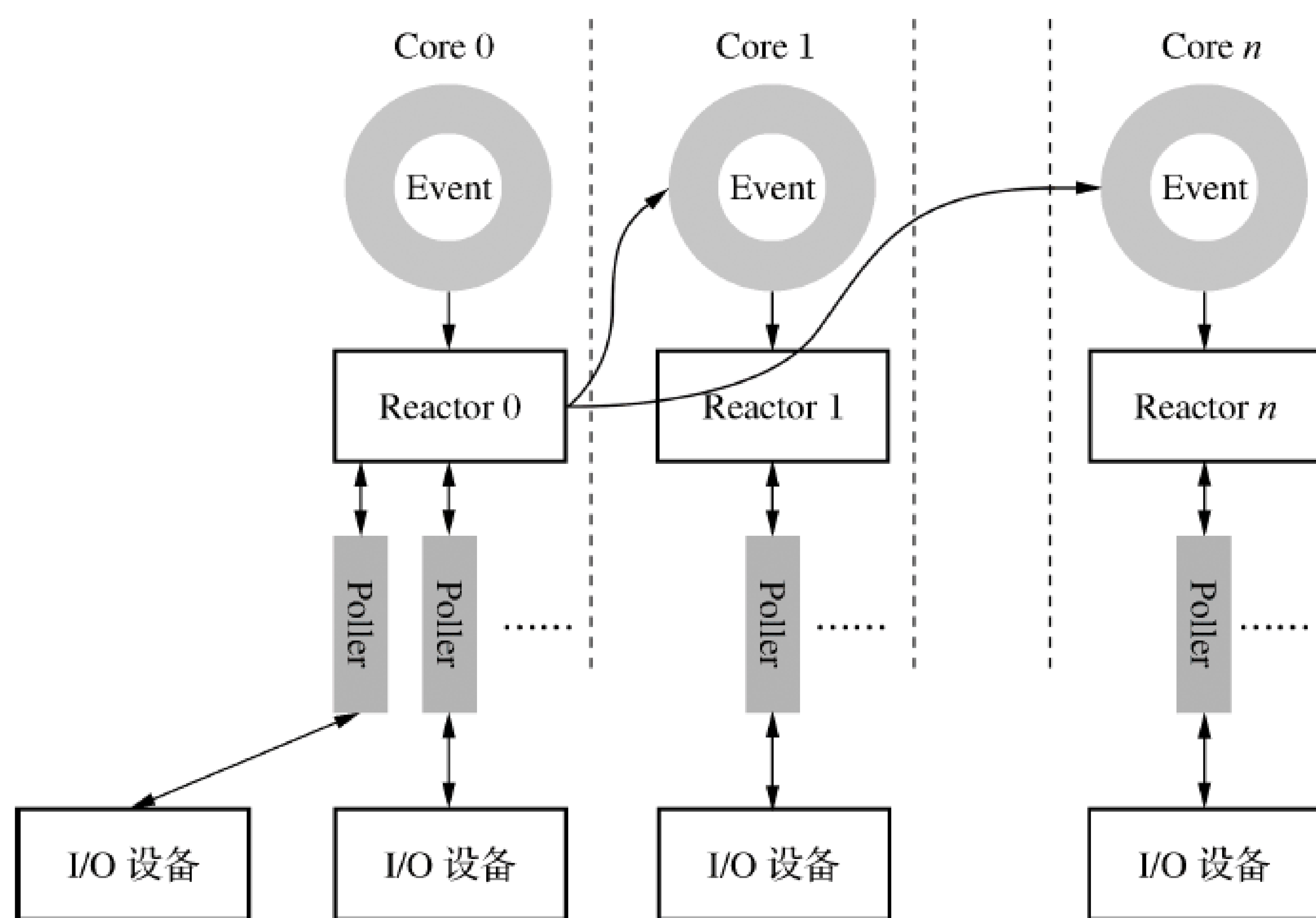


图 22 - 57　SPDK 框架中的线程模型

3）I/O 处理无锁化机制

➢ SPDK 的 I/O 采用 RTC 处理模型，核心思想是让一个线程执行完所有的任务从而避免线程切换、上下文切换和缓存数据换出。
➢ 在 I/O 路径上也采用了无锁化机制，SPDK 提供了用于 Reactor Thread 与 I/O 设备映射的 I/O Channel 结构，不同的 Reactor Thread 操作同一个 I/O 设备时拥有不同的 I/O Channel。
➢ I/O 设备的资源在 I/O Channel 有单独的一份副本，Reactor Thread 在访问资源时无需对设备加锁，真正需要加锁序列化访问的工作由 I/O Channel 负责，并不挤占 Reactor Thread 的时间片。

这里要强调一点，轮询机制只有在存储设备性能超过 CPU 性能且 I/O 并发量特别大的情况下才比较适合。轮询的本意是消除中断对 CPU 的打扰，但却会使 CPU 处于 100% 的运行状态。试想如果存储设备性能低于 CPU 性能，为了使存储设备被充分利用却使 CPU 处于 100% 的状态，但实际发生的 I/O 量却低于 CPU 的期望值，这是不是"捡了芝麻丢西瓜"的得不偿失呢？相反，若此时采用中断机制，由于存储设备性能较低，总的中断数量不会那么高，对 CPU 的打扰也不会那么频繁，且不会使 CPU 处于满负荷状态，是不是更优的选择呢？

SPDK 也提供了原生的函数库，包括初始化与反初始化、工作线程执行、异步读写等功能函数，如下所示：

```
• spdk_app_start                          //框架运行初始化
• _spdk_reactor_run                       //Reactor Thread 执行函数
• spdk_app_stop                           //框架运行终止
• spdk_nvme_ns_cmd_read                   //异步读写
• spdk_nvme_qpair_process_completions     //检查异步操作是否完成
• spdk_event_allocate                     //创建 Event
• spdk_event_call                         //传 Event 给指定的处理器核并执行 Event
• spdk_allocate_thread                    //分配 SPDK 线程
```

2. SPDK 的层次特点

SPDK 框架结构如图 22-58 所示。

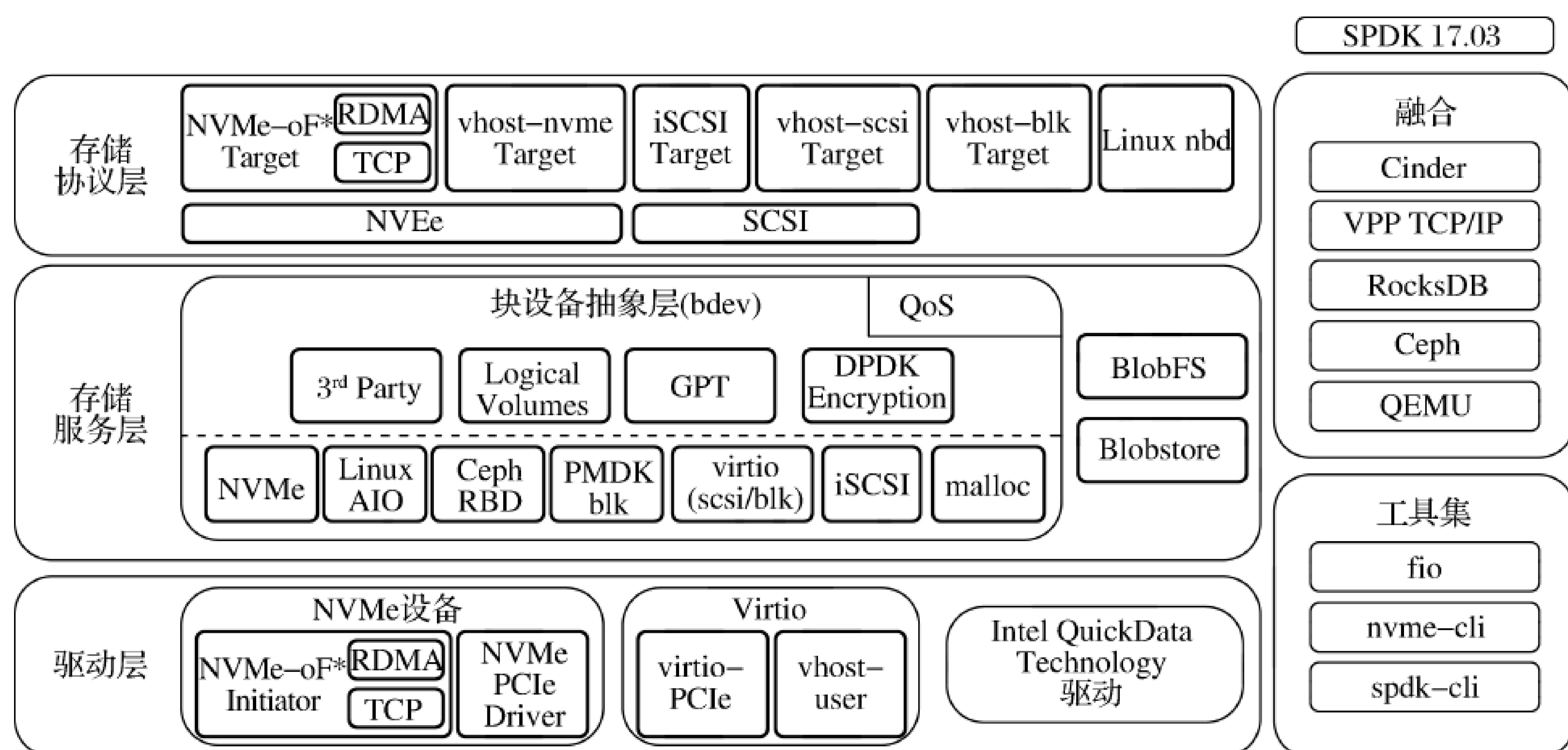


图 22-58　SPDK 框架视图(图片来自 DPDK 官网)

SPDK 框架自底向上包含了以下组件层:

➢ **驱动层(Drivers)**:包括 NVMe SSD 抽象设备和 Intel 快速 I/O(IOAT)驱动。其中 NVMe SSD 抽象设备又包括 NVMe PCI-E 驱动模块和 NVMe-oF 后端驱动模块。SPDK 的后续版本中还会对虚拟化做扩展。

➢ **存储服务层(Storage Services)**:该层又可分为两个子层,即块设备层和块设备抽象层。

- 块设备层包括了作为 SPDK 后端存储的 RBD、负责与内核设备交互的 AIO、内存分配 malloc 机制等组件;
- 块设备抽象层则包括了通用的块设备抽象 bdev 和 SPDK 实现的一个高精简的类文件语义(非 POSIX)的 Blobstore。bdev 支持连接到各种不同设备驱动和块设备的存储协议,类似于文件系统的 VFS,在块层提供灵活的 API 用于额外的用户功能,例如磁盘阵列、压缩和去冗等。

➢ **存储协议层(Storage Protocols)**:包括 NVMe-oF Target 和 iSCSI Target 等,其中 Target 表示存储设备。

本章小结

本章着重介绍了以SDN、NFV等为代表的新一代通信技术以及相关数据面加速框架。

软件定义网络部分着重介绍了SDN相关的框架、协议、链路发现机制和开源项目等。同时也介绍了基于SDN的网络功能虚拟化(NFV)等技术体系与架构。最后还介绍了5G时代非常常用的网络切片相关技术。

在数据面加速技术部分则介绍了两种基于Intel平台的I/O加速框架,即基于网络I/O的DPDK加速框架和基于存储I/O(NVMe SSD)的SPDK框架。

第23章　应用软件网络协议栈

协议栈对于大多数的软件系统是很重要的组成部分，是处于应用软件底层的“基座”。这是因为大多数软件都具有通信功能，有通信就会有通信标准和协议。而在现有的协议体系中，以 TCP/IP 协议簇最为通用和知名。在协议体系中，一般不可能由一种协议搞定所有的通信事务，因此协议是按照分层的结构“堆叠”起来共同发挥作用的，这也是“协议栈”这个名词的由来。

OSI 定义了从物理层到应用层的七层参考模型，由这些层对应的协议构成了 OSI 协议栈（例如应用层的 HTTP 协议既可以在传输层的 TCP 协议上运行也可以在 UDP 协议上运行，而无论 TCP 还是 UDP 都只能运行于网络层的 IP 协议之上），而 TCP/IP 则只定义了从物理层到应用层的五层模型，如图 23－1 所示。操作系统通过协议栈驱动的方式实现了 OSI 参考模型的下半部分，上半部分则一般由应用软件或中间件实现。

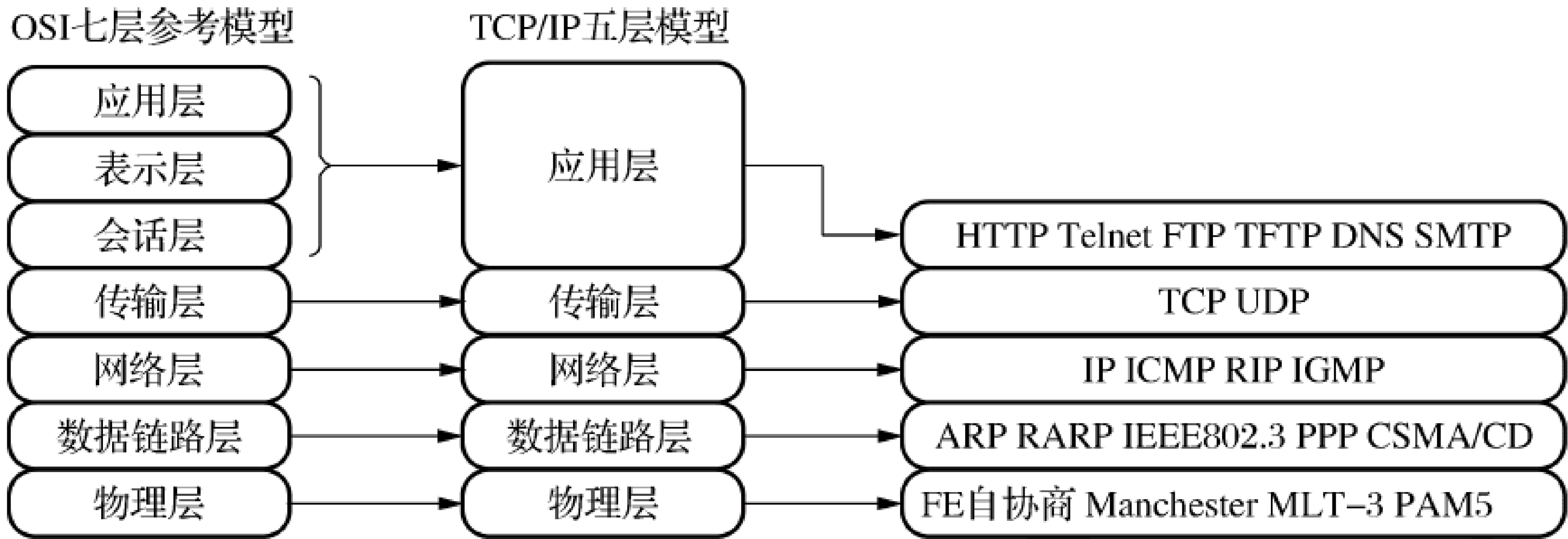


图 23－1　OSI 七层参考与 TCP/IP 五层模型的对应关系

在本章中，我们将阐述 TCP/IP 五层模型的网络层、传输层和应用层中特别是与视联网与物联网有关的各种协议。这些协议有的具备堆叠关系（例如 HTTP over TCP），有的则不具备堆叠关系。这里我们并不准备阐述所有的协议，对于常见的耳熟能详的协议和物理层、链路层协议将予以略过，对于不常见的或新出现的协议将重点描述。

本章将按照图 23－2 和图 23－3 所示的提纲介绍网络层协议和传输层协议。在应用层协议部分，将分成视联网协议栈和物联网协议栈两部分予以介绍。

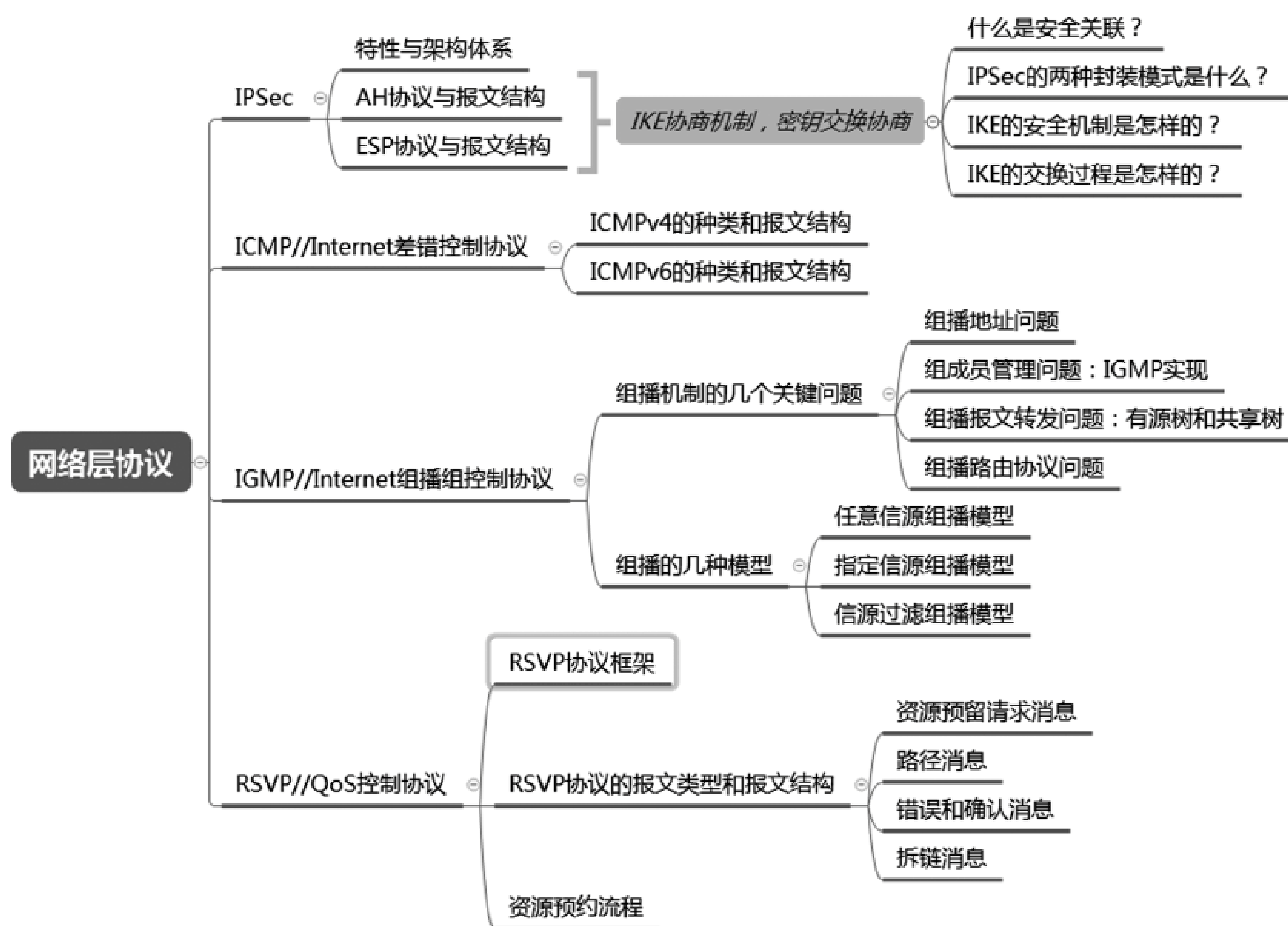


图 23－2　网络层协议部分的提纲

- 传输层协议
  - SCTP
    - 什么是SCTP？与TCP有什么不同？
    - SCTP的特征
      - 多宿主
      - 多流
      - 初始化保护
      - 消息分帧
      - 平滑关闭
      - 选择性确认
      - 自动保活
    - SCTP的报文结构
    - SCTP的握手协商流程
  - QUIC
    - 什么是QUIC协议？与TCP、TLS有什么不同？
    - QUIC的协议特征
      - 建立连接的延时很低
      - 改进了拥塞控制算法
      - 基于Stream和Connection级别的流量控制
      - 没有队头阻塞的多路复用
      - 数据可加密
      - 连接可迁移
      - 前向纠错
    - QUIC的报文结构
    - QUIC的握手协商流程，区分两种情况
      - 客户端已知服务器参数
      - 客户端未知服务器参数
  - UDT
    - 什么是UDT协议？适用于什么样的网络？解决了TCP的什么问题？UDT的优势在哪？
    - UDT的软件架构（UDT软件栈）
    - UDT的接收与发送机制
    - UDT的连接与关闭机制，基于两种模式
      - C/S模式
      - 会合连接模式
    - UDT的定时器机制
    - UDT的多路复用机制
    - UDT的报文结构
  - SSL/TLS/DTLS
    - TLS的逻辑架构以及在协议栈中的位置
    - TLS的握手协商流程
    - TLS的身份认证与密钥协商
    - DTLS握手协商流程
  - TCP相关技术
    - TCP的拥塞控制机制
      - 拥塞控制的相关名词解释
      - 拥塞控制的几个阶段：慢启动、拥塞避免、快速重传、快速恢复
      - 拥塞控制状态机
      - 随机早期检测
      - 拥塞控制的各种算法：基于丢包、基于时延、基于链路容量、基于自学习
      - BBR算法
        - 针对传统TCP在长肥网络中的劣势提出了改进
        - BBR拥塞控制的几个阶段：慢启动、拥塞避免、稳定运行
        - BBR算法的工作机制
    - TCP加速技术
      - ZetaTCP加速技术的特点
      - mmTrixTCP加速技术的特点
      - （*都是基于自学习的单边拥塞控制算法*）

图 23－3　传输层协议部分的提纲

23.1　网络层协议

网络层协议(IP 层)是通过 IP 来寻址以建立两个节点之间连接的协议,它为从源端(发送端)的传输层投递过来的报文分组选择合适的路由节点并投送到目的端的传输层。因此,网络层协议应包括以下两个任务:

- **层间接口**:网络层将传输层的报文封装成 IP 分组/包并向下层协议传递。
- **路由接口**:选择从源端到目的端的合适传输路由。

网络层的具体协议如表 23－1 所示(包括但不限于)。

表 23－1　常见的网络层协议

协议分类	协议类型	协议解释	协议说明
报文承载协议	IPv4/IPv6	Internet Protocol	互联网通信协议
	IPSec	Internet Protocol Security	互联网安全协议
互联网控制协议	ICMPv4/ICMPv6	Internet Control Message Protocol	互联网控制报文协议
	IGMP	Internet Group Management Protocol	互联网组管理协议
	RSVP	Resource Reservation Protocol	资源预留协议
地址解析协议	ARP	Address Resolution Protocol	地址解析协议
	RARP	Reverse Address Resolution Protocol	反向地址解协协议
自治系统间路由协议	BGP	Border Gateway Protocol	边界网关协议
自治系统内路由协议	OSPF	Open Shortest Path First	开放式最短路径优先协议
	RIP	Routing Information Protocol	路由信息协议
	IS-IS	Intermediate System－to－Intermediate System	中间系统到中间系统协议

23.1.1　IPSec

IPSec 是由互联网工程任务组(IETF)发布的,是为 IP 业务提供安全保护的协议标准族(IPSec 不仅仅是一个协议)。虽然 IPSec 在 IPv4 中是可选项,但在 IPv6 中却是必选的支持项,这也可以看出网络层对于安全的日益重视。IPSec 具有以下安全特性:

- 发送端可以对网络包进行加密,保证了数据机密性(Data Confidentiality)。
- 接收端可以对接收到的网络包进行认证,确保其中途未被篡改,保证了数据完整性(Data Integrity)。
- 接收端可以认证发送端是否合法,防止中间人攻击,保证了数据来源可靠性(Data Authentication)。
- 接收端可以检测和拒绝接收那些过时和重复的网络包。

➢ 接收端可以通过序列号机制保证防重放(Anti-Replay)特性。

IPSec 广泛应用于三层数据加密领域,例如 IPsec VPN。基于 IPSec 的三层安全加密具有以下优势:

➢ 支持密钥自动协商(Internet Key Exchange,IKE,互联网密钥交换)机制;

➢ 与普通 IP 协议完全兼容,基于 IP 协议的应用无需修改代码即可适配 IPSec;

➢ 以 IP 包为单位封装数据,避免了 TCP 流的边界问题,封装灵活。

IPSec 是一种三层包的加密机制,因此 IPSec 协议与 IP 协议一样同处于第三层。但凡是加密必然伴随解密,因此在封装 IPSec 包之前需要对包体内容的加解密密钥、认证机制等进行协商。在 IPSec 架构下定义了两种协议,即 AH(Authentication Header,认证报文头)协议和 ESP(Encapsulating Security Payload,封装安全载荷)协议,分别应用于包的认证和包的加密封装。协商是采用 IKE 机制完成的。

基于 AH 和 ESP 两种协议的 IPSec 体系架构如图 23-4 所示。

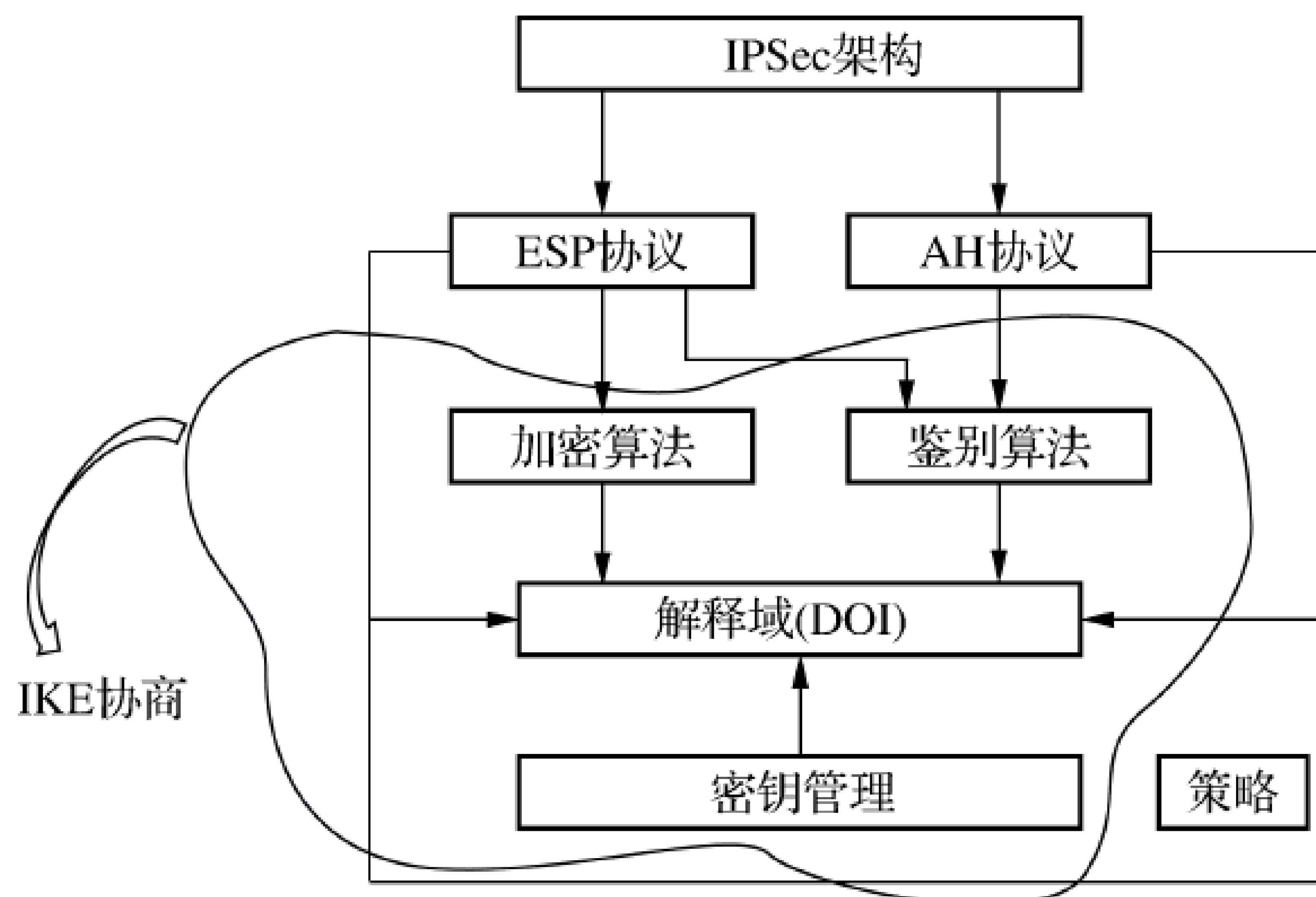


图 23-4　IPSec 体系架构

1. AH 协议

AH 是通过在 IP 报文头和 IP 包体中间插入一个认证摘要头实现的,也就是在每个 IP 报文头后面都插入了身份验证的报文头。因此 AH 协议不能简单理解为一个四层协议。

在 IP 报文头(如图 23-5 所示)中,AH 协议(AH 摘要头)的 8 位协议号是 51。AH 协议的作用是提供数据源认证、完整性校验和防止报文重放。不过虽然 AH 可以防止篡改,却不能抵御窃听,因为 AH 协议并不负责报文体的加密。

所谓摘要,就是将任意长度的报文通过散列算法变换成固定长度的字符串,使该字符串不再具有散列之前的明文特征。由于散列算法是不可逆的,因此理论上是不能由字符串逆向推导出明文的,且明文中哪怕只改了一个字符,其摘要都会变化(相同散列算法下的相同明文的摘要相同),因此多用于数据摘要和认证的场景。

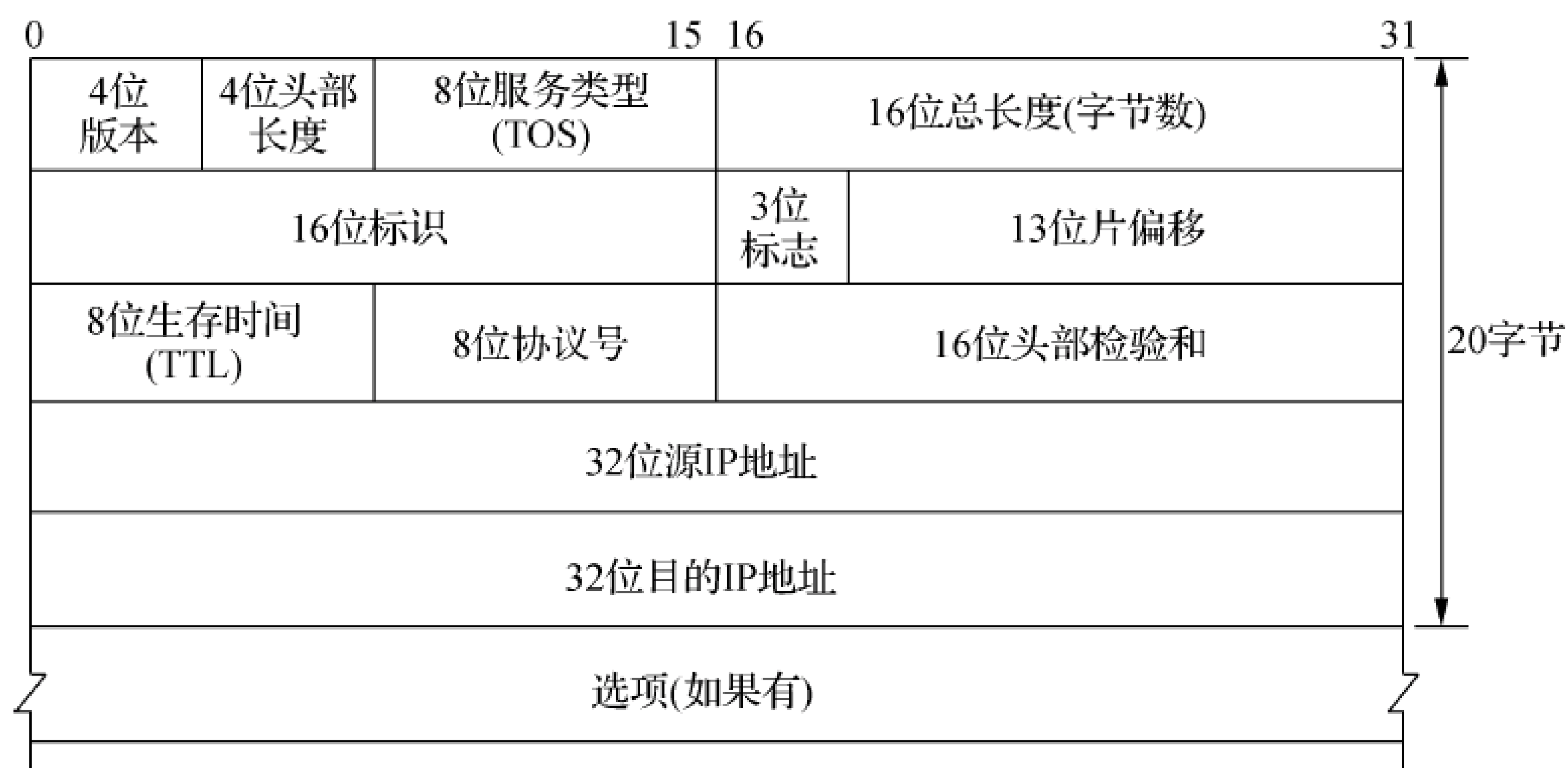


图 23－5　IP 报文头结构

在 AH 协议中常用的摘要算法包括 MD5、SHA1 等，其中 MD5 算法计算速度快但安全程度低，SHA1 则正好相反。AH 协议是对整个 IP 包进行 Hash 摘要（如图 23－6 所示），AH 头部结构如图 23－7 所示。



图 23－6　AH 在 IP 包中的位置（传输模式下）及摘要的范围

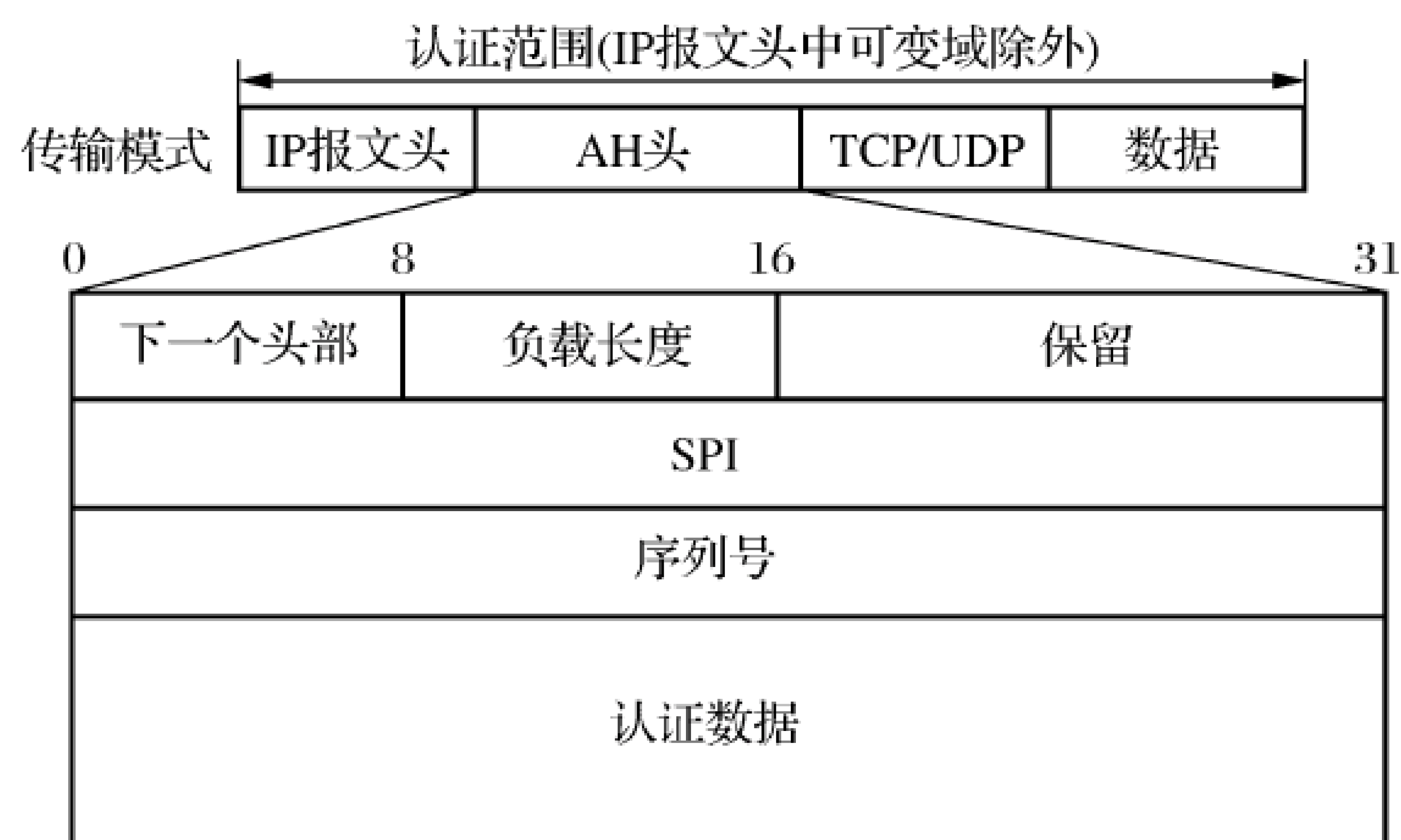


图 23－7　AH 头的结构（传输模式下）

2. ESP 协议

ESP 是一种能够实现数据加密、数据源认证、完整性校验和防重放等多种功能的协议，其在 IP 报文头中的 8 位协议号是 50。如图 23－8 所示，ESP 的封装过程是这样的：

（1）对原始 IP 包进行加密（ESP 协议中的加解密一般采用对称算法以降低计算强度）。

（2）在密文的前面创建一个 ESP 头，其结构如图 23－9 所示。

（3）将上述两部分数据（原始 IP 包＋ESP 头）进行摘要（MD5 或 SHA1 算法）并放入密文和填充区之后。

(4) 将上述报文作为新的 IP 报文体替换到原始 IP 报文头的后面。

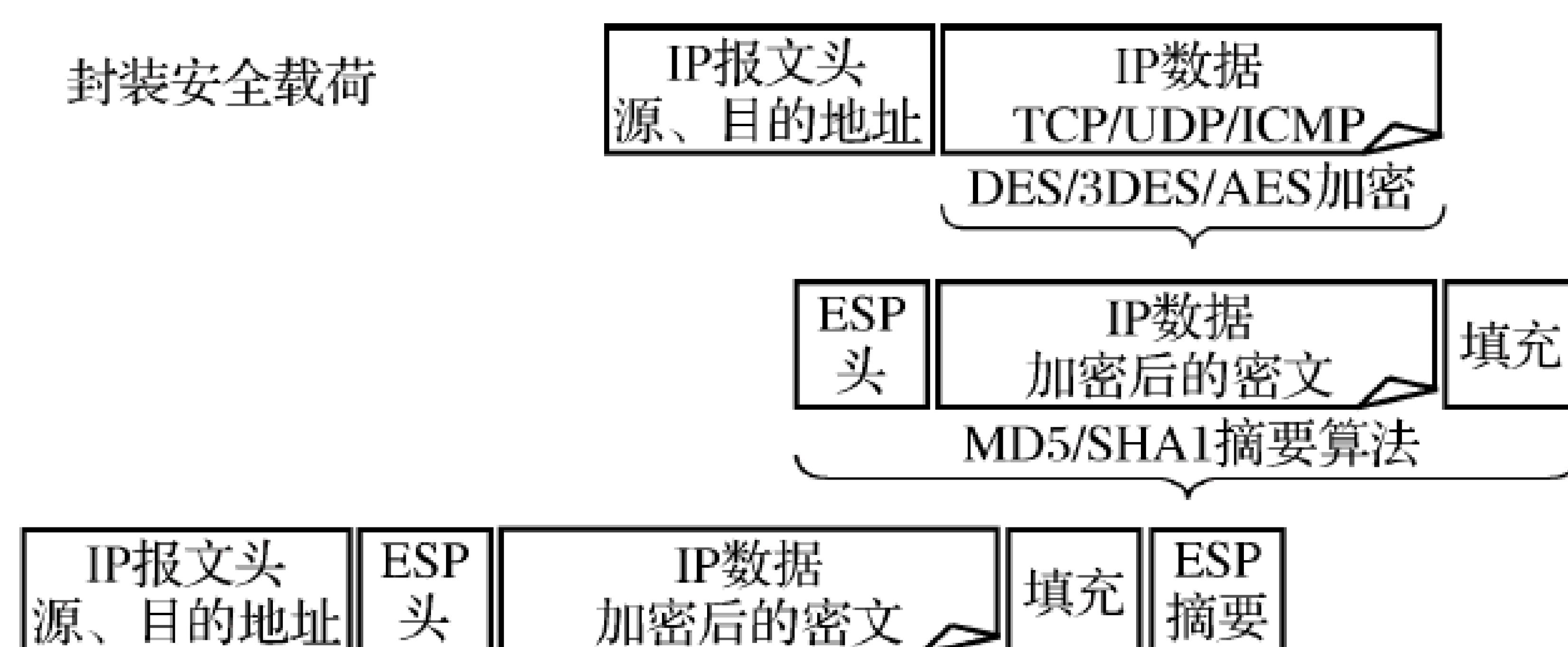


图 23－8　ESP 封装过程(传输模式下)及摘要范围

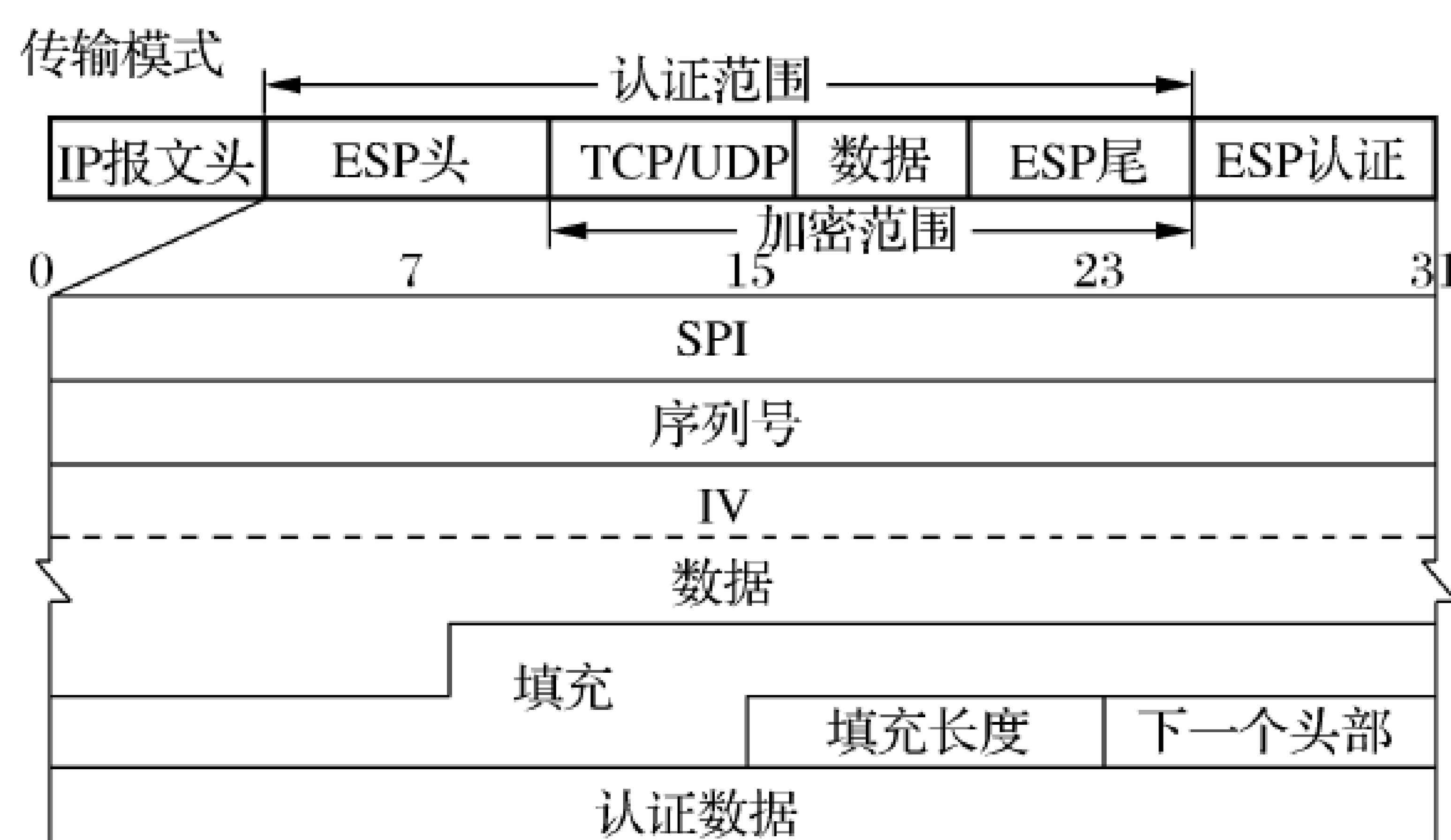


图 23－9　ESP 头的结构(传输模式下)

从上述流程可以看出,ESP 与 AH 的最大不同就是 ESP 对要保护的数据进行加密后再封包。常用的加密算法包括 DES、3DES、AES 等,算法的安全性从高到低依次为 AES→3DES→DES。在实际运用时,ESP 往往与 AH 联合使用,比如先对报文进行 ESP 封装,再对封装后的报文进行 AH 封装,故而报文体从内到外的顺序是:原始 IP 报文(报文头＋报文体)→ESP 头→AH 头→原始 IP 报文头,如图 23－10 所示。

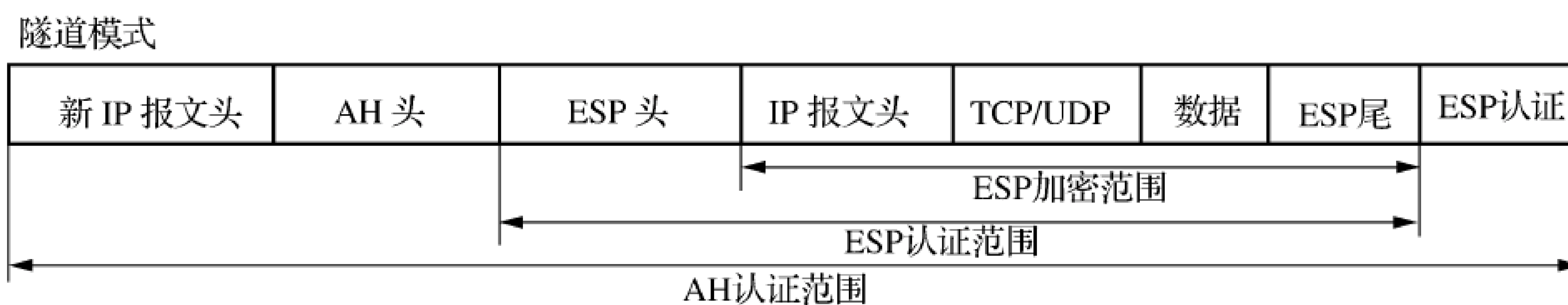


图 23－10　AH 与 ESP 组合使用的实例

表 23－2 列出了 AH 协议与 ESP 协议的性能对比。

表 23－2　AH 协议与 ESP 协议的性能对比

安全性	AH	ESP
协议号	51	50
数据完整性校验	支持	支持(不验证 IP 头)

续表 23－2

安全性	AH	ESP
数据源验证	支持	支持
数据加解密	不支持	支持
防重放	支持	支持
NT-T(NAT 穿透)	不支持	支持

3. IKE 协商机制

在 IPSec 架构中无论是 ESP 需要的加解密算法，还是 AH 需要的认证摘要算法，都需要通信双方进行协商才能正常使用。IKE(互联网密钥交换)就是这样一种支持交换协商的协议，它为 IPSec 提供了自动的密钥交换协商和建立安全关联(Security Association，SA)的服务，并且将协商好的参数和密钥上交给 IPSec。因此，IKE 作为一种基于 UDP 的应用层协议，也是 IPSec 的参数协商机制。

1) 安全关联

所谓安全关联(SA)是一个抽象的概念，是为 IPSec 提供安全数据流的单向逻辑关系，是对等的通信双方对某些要素的约定，这些要素包括协议的封装模式、加解密算法、使用 AH 还是 ESP 协议等。SA 就像一个代理桥头堡一样，通信双方各需要一个 SA 用于代理自己。

SA 也与安全协议的种类有关系，例如若同时使用 AH 和 ESP 两种协议，则通信的每一方会具有两个独立而嵌套的 SA 来代表 AH 和 ESP 及其嵌套关系，所有流经同一个 SA 的数据流都会得到同样的服务。SA 可以基于手工配置方式，也可以基于 IKE 自动协商方式。

一个 SA 由以下元素的联合体唯一标识：

- 安全参数索引(Security Parameter Index，SPI)：这是个位于 AH 和 ESP 头域中的 32 位的整型数字，用于接收端识别数据流与 SA 的绑定关系；
- 目的端 IP 地址；
- 安全协议号(IP 报文头中对于 AH 或 ESP 的标识)。

这里还要提及两个数据库：**安全关联数据库(Security Association Database，SAD)**和**安全策略数据库(Security Policy Database，SPD)**。前者用于存放与 SA 关联的状态数据的数据结构，后者指明 IP 包所对应的安全服务及其服务的获取方法。

2) 封装模式

在 IPSec 的穿透过程中存在两种封装模式，即隧道模式和传输模式，如图 23－11 所示。

- **隧道模式(Tunnel)**：使用整个 IP 包来计算 AH 或 ESP 头，AH 或 ESP 头＋ESP 加密的原 IP 报文体数据被封装在一个新的 IP 包中，如图 23－12 所示。隧道模式相当于 IP 包的嵌套封装，具有两层包头，一般用于穿透互联网或两个网关之间的通信场景。

协议＼模式	传输模式	隧道模式
AH	IP AH 数据	IP AH IP 数据
ESP	IP ESP 数据 ESP-T	IP ESP IP 数据 ESP-T
AH-ESP	IP AH ESP 数据 ESP-T	IP AH ESP IP 数据 ESP-T

图 23－11　两种封装模式在 AH 和 ESP 协议的比较

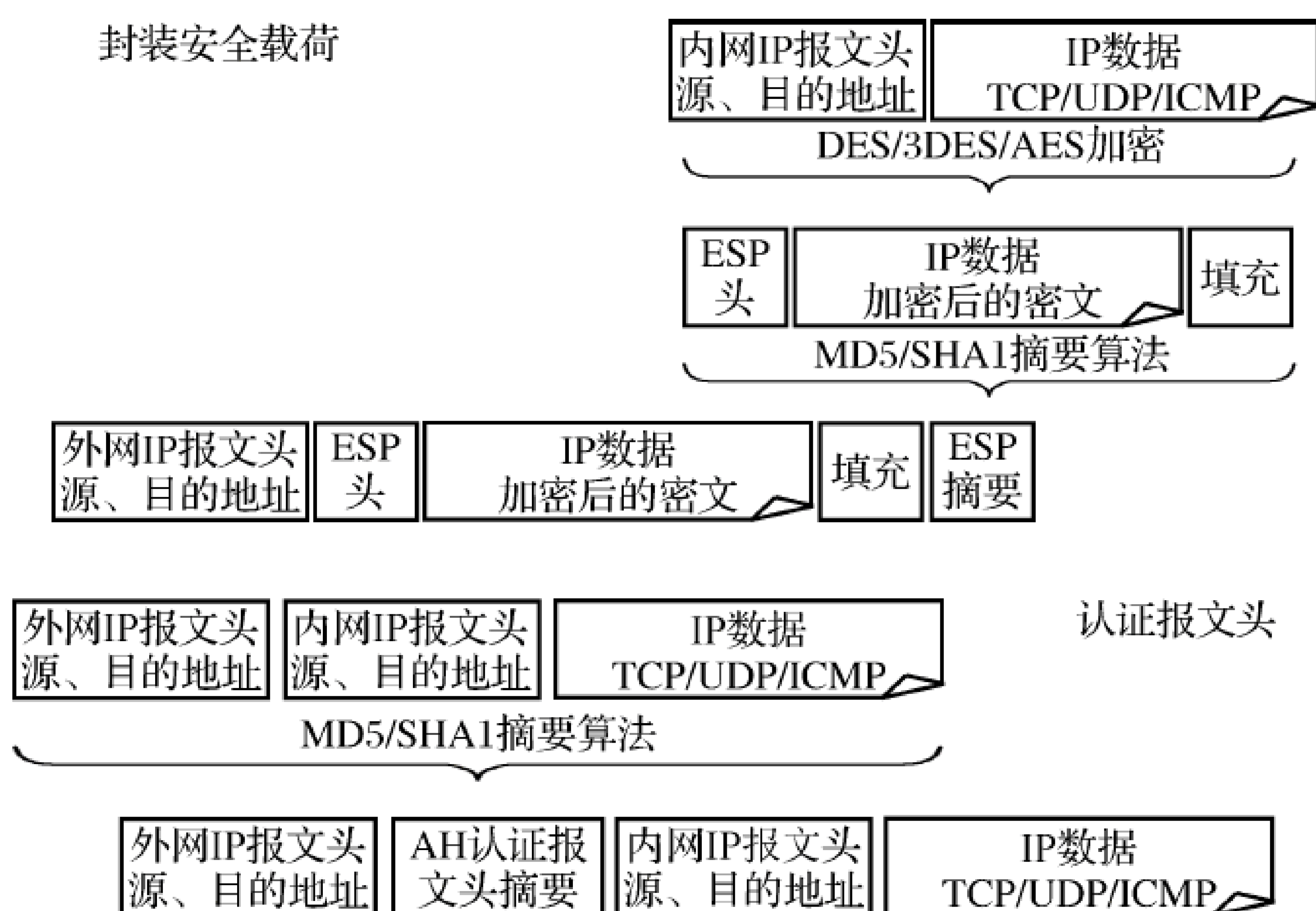


图 23－12　隧道模式中 ESP 与 AH 的封装过程

➢ **传输模式(Transport)**:使用传输层的数据计算 AH 或 ESP 头,AH 或 ESP 头＋ESP 加密的原 IP 报文体数据替换原来的 IP 包报文体数据。一般用于局域网、主机到主机或主机到网关的通信场景。

隧道模式在应用场景中更为常见,因为增加了一层 IP 报文头封装,虽然在解析的时候多了一层开销,但却可以适用于从 PC 到 PC 乃至网关到网关的所有通信场景,也非常适合穿透互联网的通信场景。而传输模式中的数据包由于只含有原 IP 包的目的地址,因此在穿透时很可能被丢弃。

3) IKE 的安全机制

IKE 不是在网络上直接传输密钥,而是通过一系列数据的交换,最终计算出双方共享的密钥,因此采用 IKE 机制不怕密钥被截获。IKE 具有以下安全机制:

➢ **数据认证**:数据认证包含两方面的内容,即身份认证和身份保护。

- 身份认证支持预共享密钥(pre-shared-key)认证和基于 PKI 的数字签名认证两种方式。
- 身份保护支持身份数据在密钥产生之后再加密传送。

➢ **交换及密钥分发算法 DH(Diffie-Hellman)**:通信的双方可以在不传输密钥的情况下

通过交换其他类型的数据计算出共享的密钥，即由其他类型数据推导出密钥。

- DH 算法的复杂度很高，即使数据被截获了截获者也计算不出真正的密钥，因此保密性很强。
- DH 算法是 IKE 安全机制的核心。

➢ **完善的前向安全性（Perfect Forward Secrecy，PFS）**：所谓前向安全性，是指一个密钥即使被破解了也不影响其他密钥的安全性，因为密钥之间没有派生关系。前向安全性是由 DH 算法保障的。

4）IKE 的交换过程

以 IKEv1.0 为例，IKE 的交换过程分为两个阶段。在第一阶段通信各方彼此之间要建立一个通过身份认证的、具有安全保护能力的通信通道，即建立 IKE 本身的 SA，并协商出 IKE 加密和认证的密钥；在第二阶段采用第一阶段建立的安全隧道为 IPSec 协商安全服务，也就是为 IPSec 协商具体的 SA 和密钥，以建立用于数据安全传输的最终的 SA。

（1）第一阶段：本阶段有两种 IKE 交换模式：主模式（Main Mode）和野蛮模式（Aggressive Mode）。主模式包含 SA 交换、密钥交换和 ID 交换验证三个步骤，其工作过程如图 23－13 所示，具体如下：

① 发送端先向对端发送本端的 IKE 策略信息，报文中携带着加密机制（DES）、散列机制（MD5-HAMC）和认证机制预共享等参数。

② 对端的应答方回复发送端。首先查找到发送端相关的策略，再将自己的信息发送给发送端，这个过程中应答方会生成自己的 Cookie 并添加到数据包中。执行完成后 SA 安全策略信息也就交换完成了。

③ SA 策略信息交换完成后，通信双方互换 DH 公共值，其中发送端的报文中还包括辅助随机数等信息。

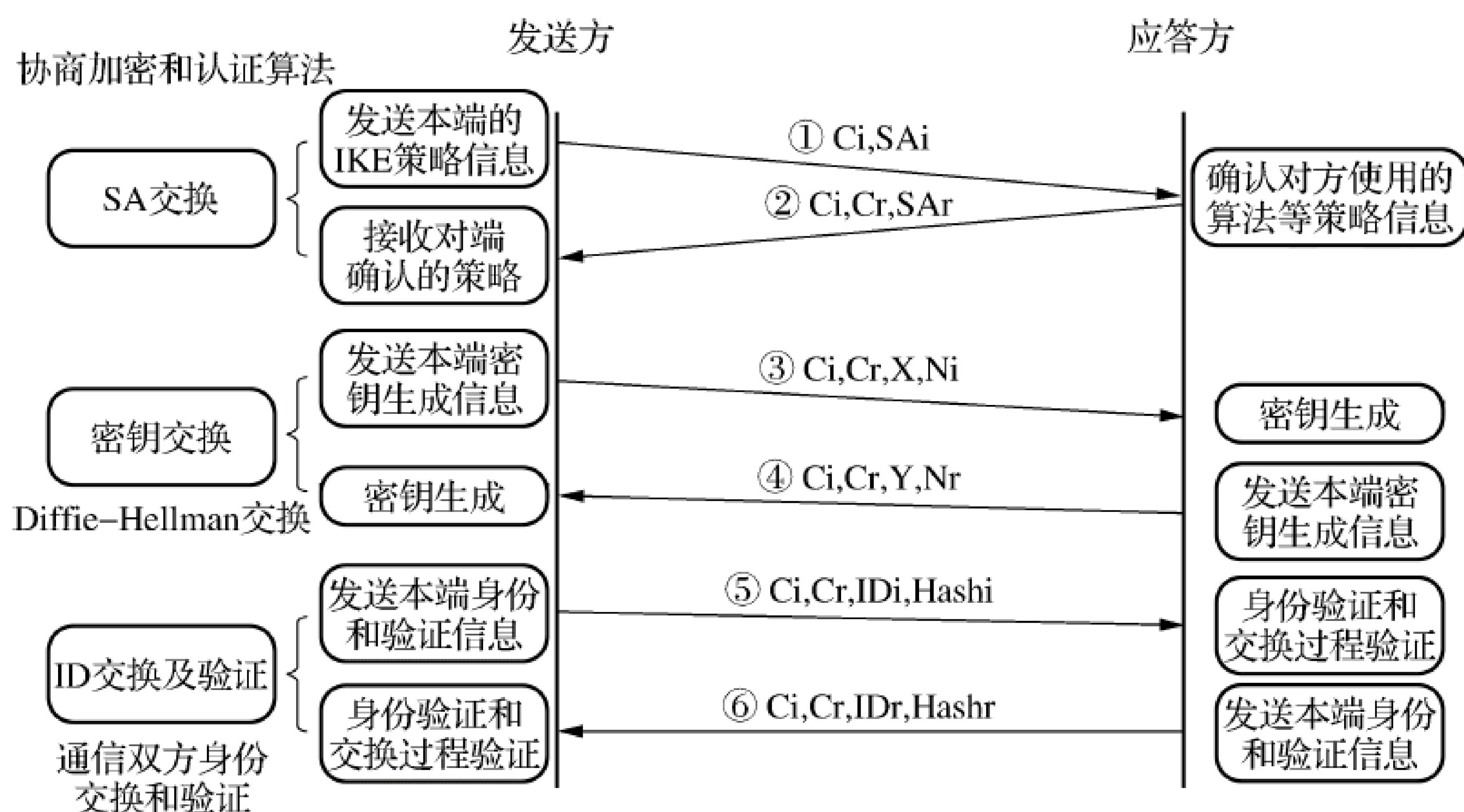


图 23－13　第一阶段主模式的工作过程

i—initiaor，发起者；r—responsor，响应者；C—Cookie；SA—安全认证；
X，Y—DH 公共值；N—Nonce，随机数；ID—己方身份信息；Hash—散列值





④ 应答方将本端的 DH 公共值和 Nonce 随机数返回给发送端,密钥就是在步骤③和④中产生的。

⑤ 发送端发送验证报文,以验证接收端就是自己要通信的对端。验证的方式可以是预共享、数字签名、加密临时值等。

⑥ 应答方也重复上一步骤,以验证发送端就是自己要通信的对端。因此,步骤⑤和⑥用于身份认证并对上述 6 步交换的内容进行总认证。

在主模式中,前面 4 个交互报文都是明文,后面 2 个是密文。野蛮模式的工作过程要比主模式简单(如图 23-14 所示),主要用于安全要求不是那么严格的场合,其步骤如下:

① 发送端向应答方发送请求,以建立 SA,发起 DH 交换。

② 应答方收到上述请求,将自己的参数和散列值返回。

③ 发起方发送请求认证应答方。

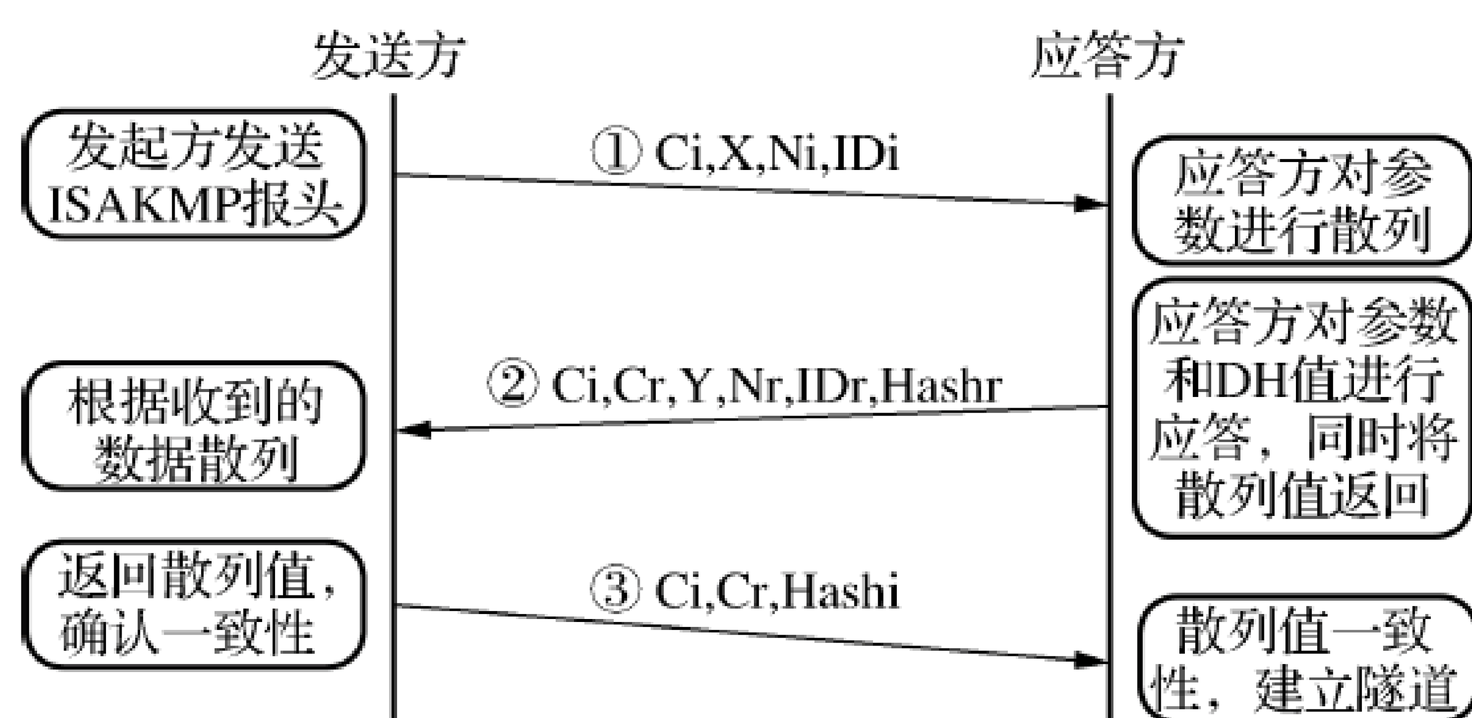


图 23-14　第一阶段野蛮模式的工作过程

ISAKMP—Internet 密钥交换协议

(2) 第二阶段:这个阶段只有一种信息交互模式,即快速模式,这种模式定义了受保护的数据连接是怎样在通信双方建立的。在这个阶段要协商安全参数以保护数据连接,并周期性地对数据连接刷新密钥信息。

IKEv2.0 简化了 IKEv1.0 的协商过程,只要一次协商就可以产生 IPSec 的密钥。

总结起来,IKE 为 IPSec 提供了自动协商交换密钥的机制以及建立 SA 的服务,并把建立的 SA 的参数和生成的密钥递交给 IPSec。IPsec 使用 IKE 建立的 SA 对 IP 报文加密或认证,同时也为 IPsec 协商密钥,供 AH/ESP 加密和验证使用,如图 23-15 所示。

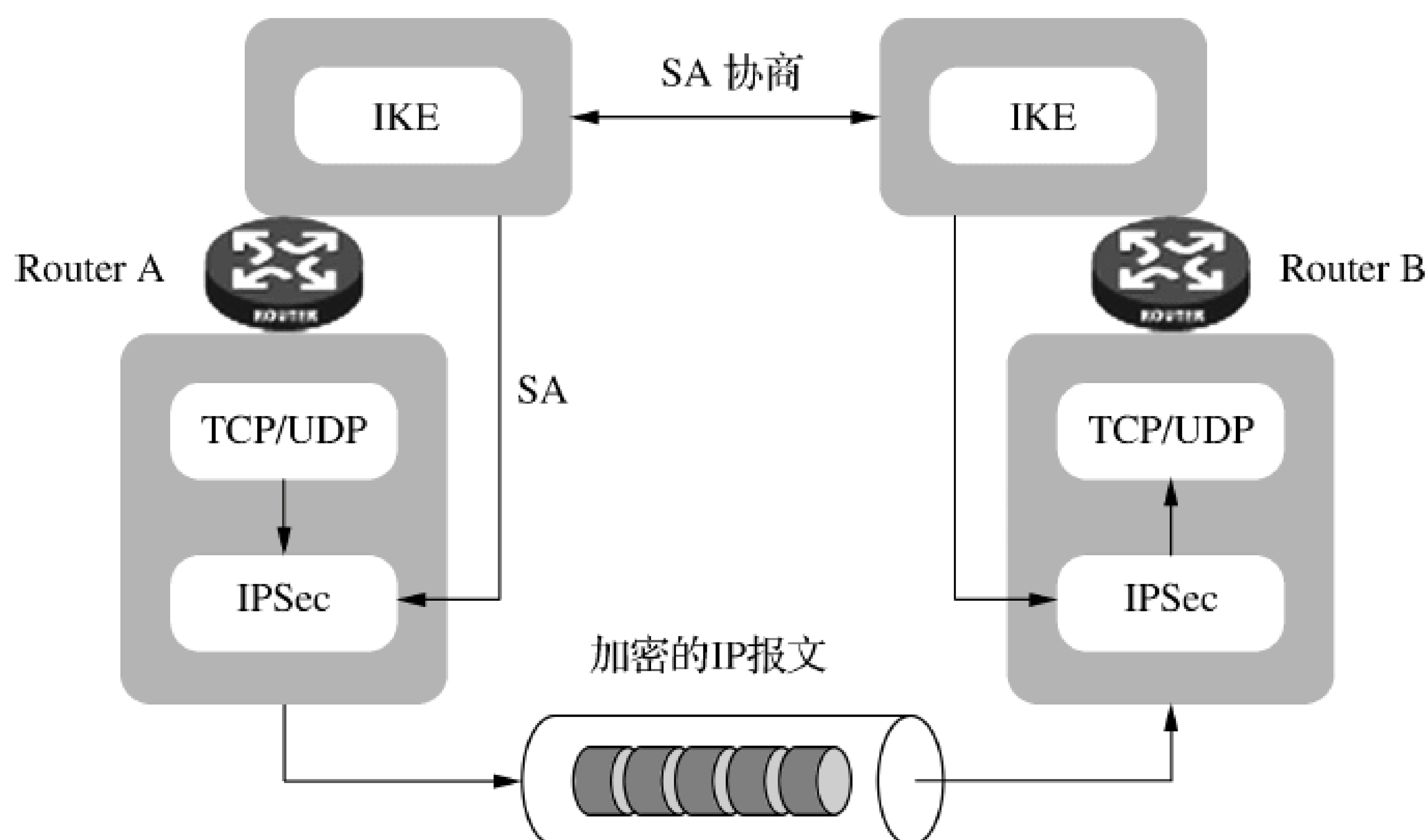


图 23-15　基于 SA 的 IPSec(图片来自 CSDN)

23.1.2　ICMP

1. ICMPv4

ICMP 是 IP 层的同层报文协议，是对 IP 协议的辅助和分担，虽然 ICMP 被 IP 协议承载，但一般也看作与 IP 协议同级。在 IP 通信过程中，由于路由器不能处理某些报文分组而不得不将这些报文分组丢弃(需要源点抑制)，或者由于目的端的 UDP 端口错误而导致目的端口不可达，这些情况都属于目的不可达错误，需要将这些信息传递到发送端。ICMP 就承担着这样的任务，它是一种面向无连接的协议。

ICMP 报文一般可以分为两类：

(1) **错误通知**：一般是由目的端或路由器向发送端发送，是用于通知出错原因的错误消息，例如源点抑制、UDP 目的端口不可达、目的端主机地址不可达等。

(2) **信息查询**：一般是发送端向目的端发送，包括 ping 命令、traceroute 命令、查询子网掩码、查询目的端时间等。

ICMP 报文的格式如图 23－16 所示。其中：

- **类型**：表示 ICMP 报文类型，它的值根据报文的内容来确定，占用 1 个字节。
- **代码**：用于确定 ICMP 报文的错误类型，例如是地址不可达还是端口不可达，占用 1 个字节。
- **校验和**：用于检测 ICMP 报文是否正确传送，占用 2 个字节。
- **标识符和序号**：不同的类型和代码具有不同的内容，不定长。图 23－16 中的标识符和序号是 ping 报文中独有的，分别占用 2 个字节。
- **选项数据**：包含于报文数据内部，用于返回出错的参数和记录出错报文的片段，帮助源节点判断错误的原因或其他问题。

IP头部 20字节	ICMP报文

<table>
<tr><td>类型(0或8)
8位</td><td>代码(0)
8位</td><td>检验和
16位</td></tr>
<tr><td colspan="2">标识符
16位</td><td>序号
16位</td></tr>
<tr><td colspan="3">选项数据(可选)</td></tr>
</table>

图 23－16　ICMP 报文格式

ICMP 报文的前三个字段(类型、代码、校验和)是通用的，每一种 ICMP 报文都有，后面的数据则随着类型的不同而发生变化，例如类型为 0 的 ping 报文就只包括了标识符和序号两部分，如图 23－17 所示。

```
⊞ Frame 343: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
⊞ Ethernet II, Src: 00:18:82:f7:d4:64 (00:18:82:f7:d4:64), Dst: 2c:44:fd:80:bc:25 (2c:44:fd:80:bc:25)
⊞ Internet Protocol Version 4, Src: 125.39.8.249 (125.39.8.249), Dst: 112.53.68.29 (112.53.68.29)
⊟ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x76cb [correct]
    Identifier (BE): 16138 (0x3f0a)
    Identifier (LE): 2623 (0x0a3f)
    Sequence number (BE): 38790 (0x9786)
    Sequence number (LE): 34455 (0x8697)
  ⊞ [No response seen]
  ⊞ Data (32 bytes)
```

图 23－17　ping 命令的 ICMP 报文实例

两类 ICMP 报文中，信息查询的最主要应用是 ping 命令，发送端通过发送 Echo Request 和接收 Echo Reply 来探测网络情况，例如 ping 报文中包含发送的请求时间，以此计算出路程来回的耗时（RTT）等。而错误通知则包含如下几类：

➢ **源抑制：**一般是由路由器瓶颈造成的，网络报文可能由于瞬时并发而造成路由器拥塞过载从而主动丢包，这时路由器应向发送端发送源抑制的 ICMP 报文以通知其降低发送速度。

➢ **目的端不可达：**包括 UDP 端口不可达、主机地址不可达、网络不可达、报文需要分段等多种情况。此时目的端主机或中途的路由器会向发送端回送这类 ICMP 报文。

➢ **超时：**网络包超过在网络中的生存时间却还没有到达，这时需要回送超时消息。路由器每转送一次数据报文就将 IP 包头部的 TTL（Time To Live，代表生存时间，一般 TTL 值是 255，不同系统的 TTL 值不一样）的值减 1，当该值减为 0 时代表已经经过了允许经过的最大路由器数，也就是超时了，这时当前路由器应沿原路回送超时消息。

➢ **路由重定向：**可以通过 traceroute 命令探测发送端主机到目的端主机的路由情况，本质上也是 TTL 的应用之一。

表 23－3 中列出了 ICMP 报文类型与代码。

表 23－3　ICMP 消息报文类型与代码

类型	代码	说明	种类
0	0	回送回答，与回送请求成对被 ping 命令使用	信息查询
3		终点不可达	错误通知
	0	网络不可达	
	1	主机不可达	
	2	协议不可达	
	3	端口不可达	
	4	需要分片，但该数据包的 DF（不要分片）已设置	
	5	源点路由选择不能完成	
	6	目的网络未知	

续表 23－3

类型	代码	说明	种类
3	7	目的主机未知	错误通知
	8	源主机是孤立的	
	9	从管理上禁止与目的网络通信	
	10	从管理上禁止与目的主机通信	
	11	对指定的服务类型网络不可达	
	12	对指定的服务类型主机不可达	
	13	主机不可达，因为管理机构已经在该主机上放置了过滤器（由 RFC 1812 追加）	
	14	主机不可达，因为主机的优先级被违背了（由 RFC 1812 追加）	
	15	主机不可达，因为主机的优先级被删除了（由 RFC 1812 追加）	
4	0	源点控制，通知发送端抑制发送数据包	错误通知
5		改变路由	错误通知
	0	对特定网络路由的改变	
	1	对特定主机路由的改变	
	2	基于指明的服务类型对特定网络路由的改变	
	3	基于指明的服务类型对特定主机路由的改变	
6	0	回送请求，与回送回答成对被 ping 命令使用	信息查询
9		路由器通告（由 RFC 1256 追加）	信息查询
	0	一般路由器通告（路由器向自己身边通告自己的存在）	
	1	不能转发一般流量（由 RFC 2002 追加）	
10	0	路由器询问（由 RFC 1256 追加）	信息查询
11		超时	错误通知
	0	传送生存时间变为了 0，被 traceroute 命令利用	
	1	规定时间内没有收到所有的分片	
12		参数问题	错误通知
	0	在 IP 包头部的某个字段中有错误或两义性	
	1	缺少所需的选项部分（由 RFC 1108 追加）	
	2	长度不对	
13	0	时间戳请求	信息查询
14	0	时间戳回答	信息查询

2. ICMPv6

基于 IPv6 的 ICMP 协议也是用于传递错误和故障信息的,但是与基于 IPv4 的 ICMP 协议相比在格式上已经发生了很大变化。与 ICMP 报文一样,ICMPv6 报文仍然分为两大类,即差错控制和信息查询。但除此之外,ICMPv6 增加了一些路由类协议功能,例如邻居发现、节点信息查询、组播监听等,也就是说,ICMPv6 实现了 IPv4 中 ICMP、ARP 和 IGMP 的功能,如表 23 – 4 所示。

表 23 – 4 ICMPv6 报文类型

类型	代码	说明	种类
1	0	目的端不可达;无路由	差错控制
	1	目的端不可达;因管理原因禁止访问	
	2	目的端不可达;未指定	
	3	目的端不可达;地址不可达	
	4	目的端不可达;端口不可达	
2	0	数据包过长	
3	0	超时;跳数已为 0	
	1	超时;分片重组超时	
4	0	参数错误;错误的包头字段	
	1	参数错误;无法识别的下一包头类型	
	2	参数错误;无法识别的 IPv6 选项	
128	0	Echo request	信息查询
129	0	Echo reply	

在 IPv6 包头中,NextHeader = 58 表示 IPv6 包头后封装了一个 ICMPv6 消息,其结构如图 23 – 18 所示,其中 ICMPv6 的报文主体长度是可变的。

0–3	4–11	12–15	16–23	24–31
版本=6	流量类别	流标签		
净荷长度			NextHeader=58	跳数限制
源地址				
目的地址				
类型	代码	校验和		
报文主体				

图 23 – 18 IPv6 包头封装的 ICMPv6 报文(图片来自 CSDN)

4 种差错控制报文的消息格式如图 23 - 19 所示。

类型=1	代码=0/1/2/3/4	校验和=ICMP校验和
未用=0		
在整个分组不超过最新IPv6 MTU(1280)情况下装载尽可能多的原始分组		

类型=2	代码=0	校验和=ICMP校验和
MTU		
在整个分组不超过最新IPv6 MTU(1280)情况下装载尽可能多的原始分组		

类型=3	代码=0/1	校验和=ICMP校验和
未用=0		
在整个分组不超过最新IPv6 MTU(1280)情况下装载尽可能多的原始分组		

类型=4	代码=0/1/2	校验和=ICMP校验和
指针		
在整个分组不超过最新IPv6 MTU(1280)情况下装载尽可能多的原始分组		

图 23 - 19　4 种差错控制报文的消息格式

除了包含 ICMP 的差错控制和信息查询功能外，ICMPv6 还具有组播收听发现和邻居发现两个功能。

（1）**组播收听发现（Multicast Listener Discovery，MLD）**：组播收听发现协议（MLD Protocol，MLDP）定义了两种 ICMPv6 报文。

- **组播收听查询报文**：组播路由器向组播组成员发送该报文以获取成员状态。
- **组播收听者报告报文**：作为对上述消息的回应，组播组成员向路由器报告自身的状态，也包括告知离开组播组（主动发送）。

（2）**邻居发现（Neighbor Discovery）**：邻居发现协议（Neighbor Discovery Protocol，NDP）定义了 5 种 ICMPv6 报文，如表 23 - 5 所示。

- **路由器通告报文**：路由器以组播方式向域内发送该通告报文，向成员告知其可用性等信息。路由器可以周期性地主动发送，也可以在收到主机的路由器请求协议后作为应答发出。
- **路由器请求报文**：主机向域内的路由器发送该报文，以获取路由器通告消息。
- **邻居请求报文**：这是一条链路层地址请求协议，用于主机验证保存在本地缓存中的链路层地址是否过期和唯一。发送的方向是从主机到邻居主机。
- **邻居通告报文**：主机在收到邻居请求消息或本机的 MAC 地址发生改变时，采用该协议向邻居主机进行通告（新的 MAC 地址）。包括原本由 ARP 负责的地址解析协议也由邻居请求和邻居通告报文来实现。
- **重定向报文**：路由器发送该报文以通知主机重新定向它发送分组到目的主机的路径，这意味着有一个更好的下一跳节点来替代路由器自己。

表 23-5 邻居发现协议为 ICMPv6 定义的报文类型

类型	报文名称
133	路由器请求(RS)
134	路由器通告(RA)
135	邻居请求(NS)
136	邻居通告(NS)
137	重定向消息

23.1.3 IGMP

组播机制解决了单播机制下因数据重复拷贝而重复占用网络带宽的问题,也解决了广播机制下的报文分组洪泛而造成的带宽浪费、传输效率低下的问题。组播在电信行业有着广泛应用,例如 IPTV 就是基于组播机制实现的,IPTV 里的一个电视频道对应一个组播 IP 地址,当需要接收该频道的信息时就加入该组播组,否则就离开。组播的汇聚点叫作中介点(Rendezvous Point,RP),组播将数据报文单播发送到 RP,RP 将这些报文的拷贝分发给各个组播订阅者。可以看出,组播 RP 将数据报文的拷贝分发限定在传输的“最后一公里”。基于此,组播机制需要解决下面几个关键问题:

(1)**组播地址问题**:组播是向一组接收者而非全部接收者或单个接收者发送报文。组播地址问题是如何确定这组接收者的问题。一个组播组包含若干个接收者,组播组可以分为永久组播组和临时组播组两类。

(2)**组成员管理问题**:组播接收者通过加入组播组来实现对组播报文的接收,组成员管理问题是接收者怎样动态地加入和离开组播组的问题。

(3)**组播报文转发问题**:组播报文在网络中怎样被转发到组播接收者的问题。

(4)**组播路由协议问题**:就是组播转发树(组播报文的转发路径)是如何构建的问题。

我们先介绍一下组播模型的分类。

- **ASM 模型**:Any-Source Multicast,任意信源组播模型。任意一个发送者都可以作为组播源向某组播组地址发送信息。接收者无法预先知道组播源的位置,但可以在任意时间加入或离开该组播组。
- **SSM 模型**:Source-Specific Multicast,指定信源组播模型。组播组的成员可能对某些组播源发送的报文分组感兴趣,而忽略其他源发送的报文分组。该模型为用户提供了一种能够在客户端指定组播源的传输服务,接收者事先通过其他手段预先知道了组播源的具体位置。
- **SFM 模型**:Source-Filtered Multicast,信源过滤组播模型。SFM 模型对 ASM 模型进行了扩展,上层软件对收到的组播报文的源地址进行检查,允许或禁止来自某些组播源的报文通过,接收者只能收到来自部分组播源的组播报文。

1. 组播地址问题

D 类网组播地址分为三种类型：预留组播地址、用户组播地址和本地管理组播地址，如图 23－20 所示。当路由器的网络层处理收到的组播数据报文时，根据组播目的地址查找组播转发表，继而对报文进行转发。

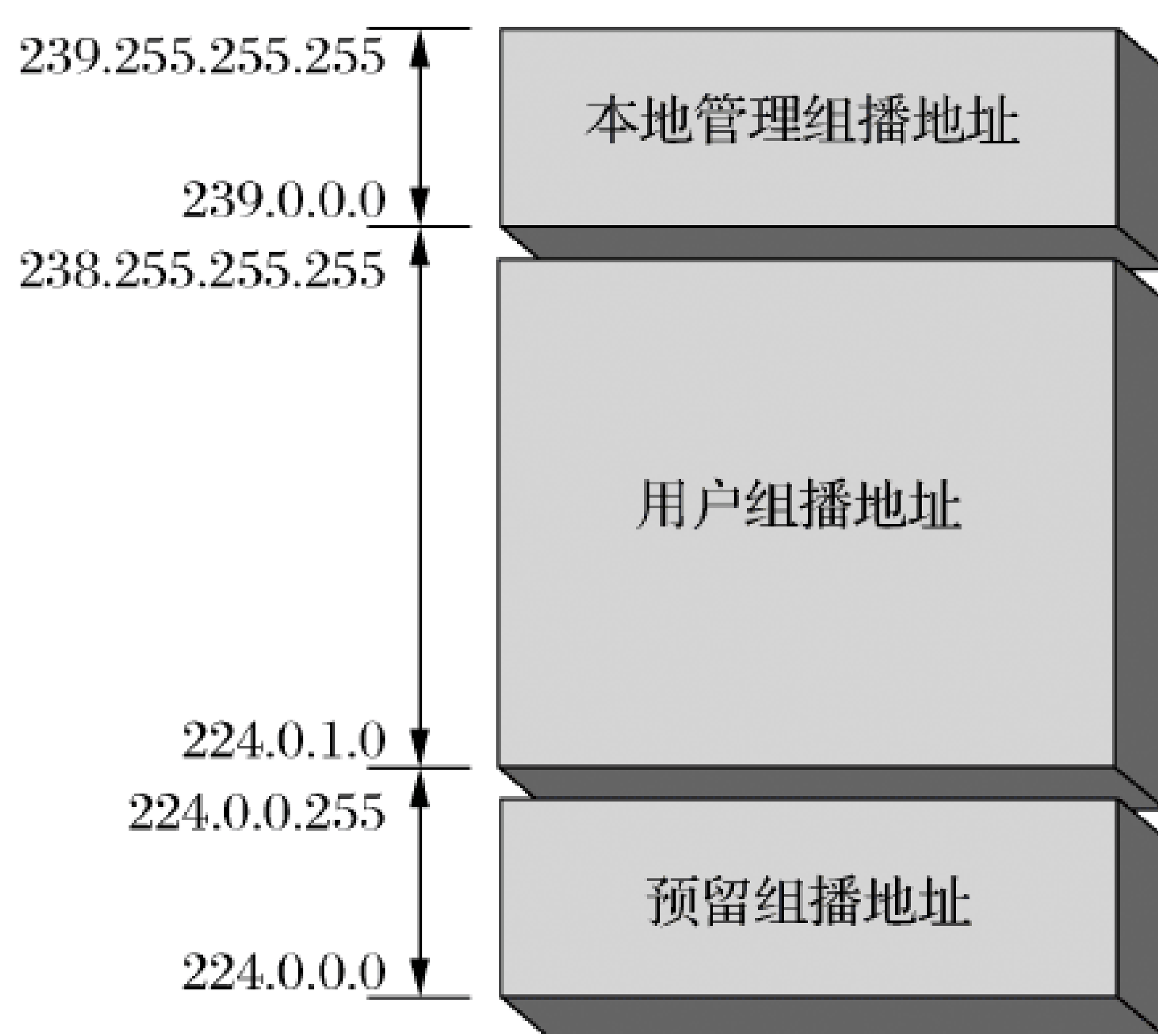


图 23－20　D 类网组播地址划分

- **预留组播地址**：范围是 224.0.0.0～224.0.0.255，这段地址被 IANA（Internet Assigned Numbers Authority，互联网数字分配机构）预留，除 224.0.0.0 外，其他地址供路由协议及拓扑查找和维护协议使用，并且只能用于局域网，且不论 TTL 为多少都不会被路由器转发。
- **用户组播地址**：范围是 224.0.1.0～238.255.255.255，全网范围内有效。其中 232.0.0.0/8 为 SSM 组地址，直接在接收者和其指定的组播源之间建立专用的组播转发路径，其余地址则为 ASM 组地址。
- **本地管理组播地址**：范围是 239.0.0.0～239.255.255.255，仅在特定的本地范围内有效。

2. 组成员管理问题

组播成员通过 IGMP（Internet Group Management Protocol，互联网组管理协议）实现加入/离开组播组。IGMP 是个组管理协议，用于在 IP 主机以及与其直接相邻的组播路由器之间建立和维护组播组的成员关系，以便路由器知道各接口所对应子网上都有哪些组的成员。

IGMP 规定了两大类功能：加入组和查询组内成员活动状态，也就是说主机既可以通过 IGMP 通知路由器希望接收或离开某个特定组播组，同时路由器也可以通过 IGMP 周期性地查询当前组播组的成员状态和成员关系信息。

截至目前，IGMP 共有三个版本：

- **IGMPv1**：定义了基本的组成员查询和报告的过程。
- **IGMPv2**：增加了组成员快速离开、查询器选举等机制。IGMP 查询器收到离开组消息后，会发送特定组查询消息来确定该组的所有组成员是否都已离开。
- **IGMPv3**：增加了组播组成员指定接收或者拒绝接收来自某些组播源报文的机制（支持指定信源组播 SSM 模型），增强了主机控制能力和查询/报告报文的功能。

所有版本的 IGMP 都支持 ASM 模型，其中 IGMPv3 可以直接应用于 SSM 模型，而 IGMPv1 和 IGMPv2 则需要在 IGMP SSM Mapping 技术的支持下才能应用于 SSM 模型。

组播协议分为三层组播协议和二层组播协议两种。三层组播协议包括组播组管理协议（IGMP）和组播路由协议（详见“组播路由协议问题”部分）。二层组播协议则包括 IGMP

Snooping 和组播 VLAN。

IGMP Snooping(Internet Group Management Protocol Snooping,互联网组管理协议窥探)是一种运行于链路层的组播协议。主机发往 IGMP 查询器的报文经过交换机时,交换机要对该报文进行监听和记录,并建立交换机端口与组播 MAC 地址之间的关系。当交换机后续再收到组播报文时根据这个关系直接向连接组播组成员的端口转发。因此 IGMP Snooping 可以解决二层环境中的组播报文泛滥问题,但要求交换机能够解读三层报文且需要对所有组播报文监听和解析。

3. 组播报文转发问题

组播通过转发树机制实现数据报文转发,有两种转发树模式:

- **有源树(Source Tree)**:以组播源为树根,将组播源到每一个接收者的最短路径结合起来构成的转发树称为有源树,也被称为最短路径树。对于每个组播组,路由器要为每个向该组发送报文的组播源建立有源树,且要保存每个组播源的路由信息,因此路由表的规模比较庞大。
- **共享树(Rendezvous Point Tree,RPT)**:与有源树不同,共享树以某个路由器(RP)作为树根,由 RP 到所有接收者的最短路径结合起来构成的转发树称为共享树。因此,所有的组播源和接收者都使用该共享树收发报文,组播源先向 RP(树根)发送数据报文,之后 RP 将报文向下转发到所有接收者,如图 23-21 所示。虽然共享树相较于有源树在路由器中保留了更少量的路由信息,但是组播源发出的报文要先经过 RP 再到接收者,这个路径可能并非最短。

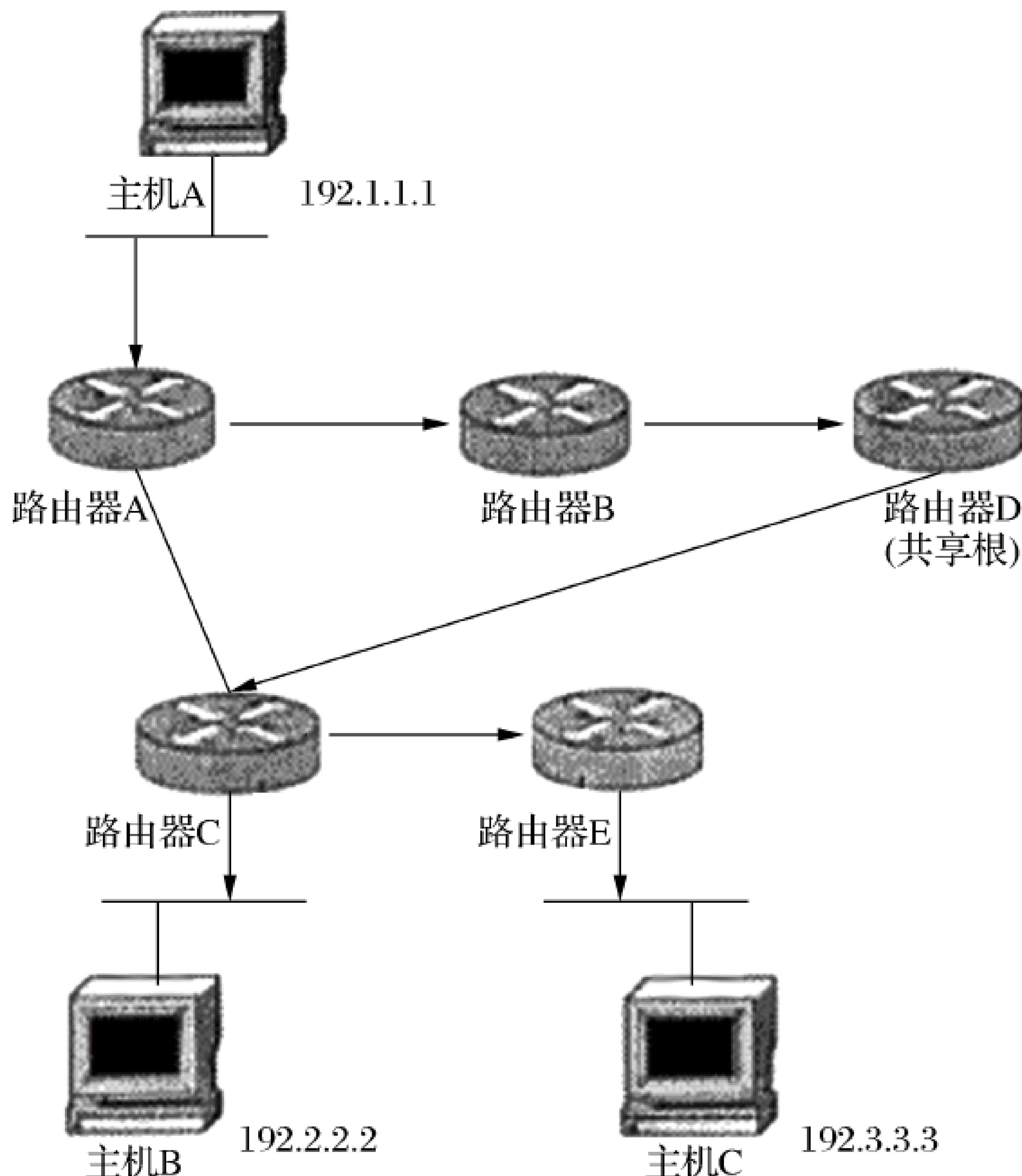


图 23-21　共享树模式实例

由于组播报文是发送给一组接收者的，因此这些接收者可以用一个组播地址标识。路由器在收到组播报文后，必须根据报文的源地址确定其正确的入接口（指向组播源方向）和下行方向，然后将其沿着远离组播源的下行方向转发。这个过程称为逆向路径转发（RPF）。路由器收到组播报文后先对其进行 RPF 检查，只有检查通过才执行转发。RPF 检查的过程为：路由器在单播路由表中查找组播源或 RP 对应的 RPF 接口，如果组播报文是从 RPF 接口接收过来的，则 RPF 检查通过，报文向下行接口转发；否则就丢弃该报文。

4. 组播路由协议问题

组播路由协议要解决转发树如何构建这个问题，即确定从组播数据源到组播组所有成员的转发树。ASM 模型的组播路由协议分为域内和域间两种，其中域内组播路由协议又可分为三大类：

- **稠密模式的组播路由协议**：稠密模式是指组播成员集中在一个小范围内，因此可以采用“洪泛 + 裁剪”的组播组进行转发。这样的协议包括 DVMRP（距离向量组播路由协议）和 PIM-DM（协议无关组播协议–密集模式）。
- **稀疏模式的组播路由协议**：稀疏模式是指组播成员分散在一个大范围内，每个特定区域内的成员较少。这样的协议包括 PIM-SM（协议无关组播协议–稀疏模式）和 CBT（基于中心的分布树协议）。
- **链路状态协议**：链路状态协议使用 SPT 向网络中的接收点发送组播数据报文。这样的协议包括 MOSPF（开放式组播最短路径优先协议）。

域间组播路由协议包括：

- **MBGP（组播边界网关协议）**：用于在自治域之间交换组播路由协议。
- **MSDP（组播信源发现协议）**：用于在 ISP 之间交换组播信源信息。

对于 SSM 模型则没有域内与域间之分，由于接收者事先知道组播源的具体位置，因此只需借助 PIM-SM 构建的通道即可实现传输功能。

23.1.4　RSVP

RSVP 是 QoS 控制协议的一种，顾名思义就是在传输的中间节点上要求为数据流保留一定的带宽资源并维护这种资源的状态。RSVP 是为 QoS 体系中的综合业务模型服务（Integrated Service）而设计的，具有以下特点：

- RSVP 采用传输接收端发起申请的策略，沿途反向申请资源。
- RSVP 本身并不处理流量控制和策略控制的参数，它仅仅是这些参数的搬运工，真正执行资源预留的实体是路径中的路由器和交换机。
- RSVP 虽建立在 IP 协议之上，但本质上还是属于三层协议。
- RSVP 是个单向协议，只能在一个方向上（接收端→发送端）预留资源。
- RSVP 可满足多个接收端的预定要求，支持这些接收端预定不同数量的资源。
- RSVP 对话由一个三元组表示：目的地址、协议号、协议端口。

➢ 为了建立并维护分组数据传输通道中各交换机的状态，RSVP 建立了异构信宿树。

RSVP 协议框架由决策控制模块、接纳控制模块、分类控制模块、分组调度模块和 RSVP 处理模块五部分构成，如图 23 – 22 所示。

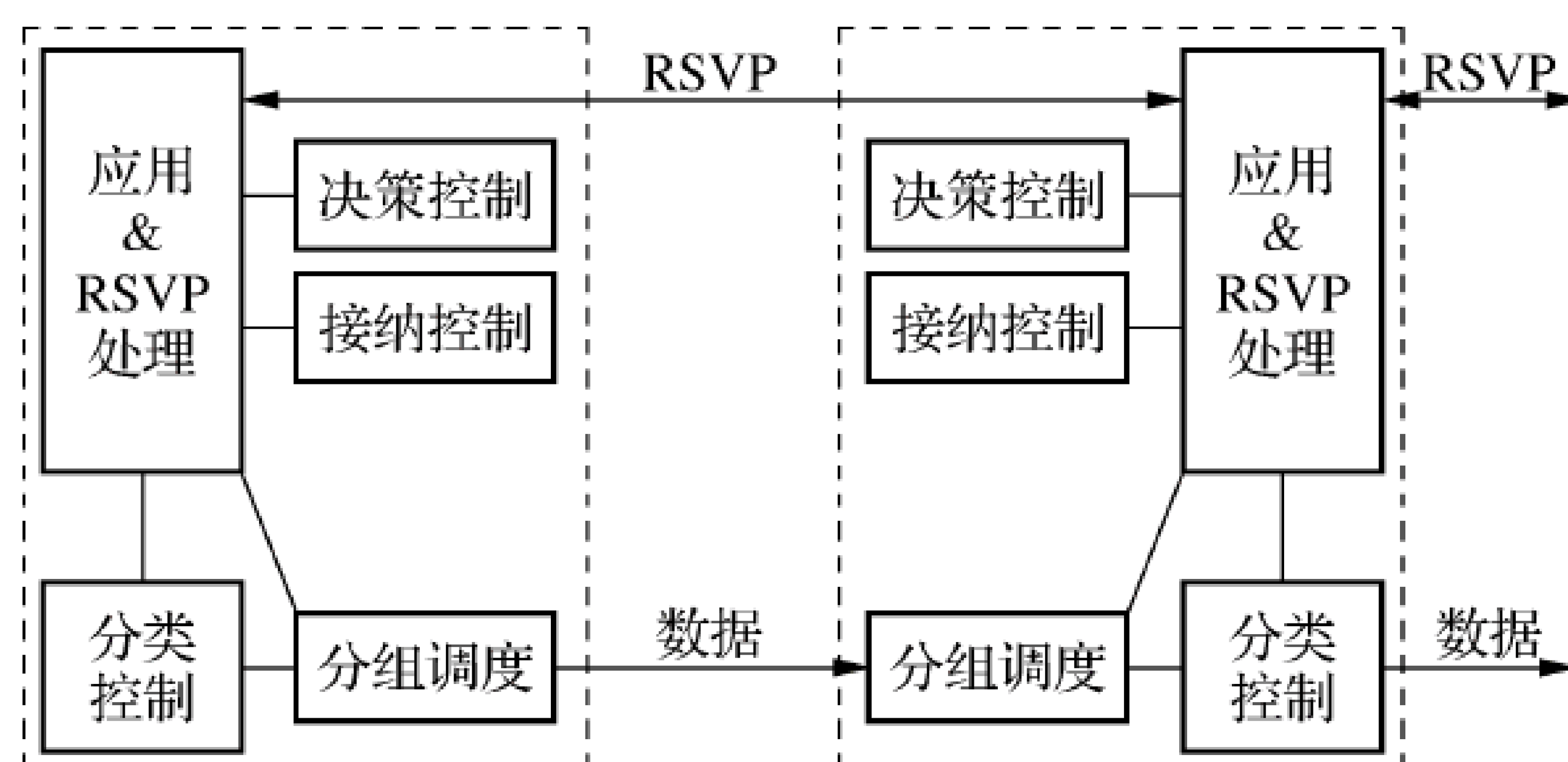


图 23 – 22　RSVP 协议框架

➢ **决策控制模块**：用于判定用户进程是否拥有资源预留的许可权。

➢ **接纳控制模块**：用于判定当前的可用资源是否能满足用户进程的需求。

➢ **分类控制模块**：用于判定数据分组的通信服务等级，实现分组过滤。

➢ **分组调度模块**：根据服务等级对数据分组进行优先级排序。

➢ **RSVP 处理模块**：根据分类和调度模块所要求通信服务质量的参数保留资源，同时也为未能获得决策控制和接纳控制许可的节点路由器产生预留错误消息并发送回接收端。

RSVP 协议包括 4 种报文消息。

1）资源预留请求消息（RESV）

网络资源预留是由 RESV 和 PATH 消息共同实现的，这类消息必须是从接收端向发送端发送，其路径是根据 PATH 消息反向推导出来的。RESV 消息想要生效必须最终到达发送端主机，只有这样才能为传输中的第一跳设置合适的 QoS 参数。

基本的资源预留请求包含一对域：流规格域和筛选规格域，用于进行流描述。

➢ **流规格域**：用于指定资源请求中的服务质量，包含了服务等级的参数和两个参量，这两个参量分别用于描述数据流的流量参量（Tspec）和定义需要保留的服务质量参量（Rspec）。

➢ **筛选规格域**：定义了数据报能否接受流规格指定的服务质量。
  - 可以根据发送端的一些特征向量来定义筛选方式，比如源 IP 地址、源端口号等。
  - 如果一个会话中的数据报不符合筛选规格，则它就不能以指定的服务质量被传送，而只能被网络以“尽力而为”的方式传送。

2）路径消息（PATH）

与 RESV 消息相反，PATH 消息是由发送端基于单播或组播的方式向接收端发送的，消息中承载了每个节点（路由器）的路径状态。RESV 消息正是通过 PATH 消息才回溯到发送端的。

3）错误和确认消息

确认消息是针对资源预留请求的确认，而错误消息则分为两类：PATH 错误消息和 RESV 错误消息。

- PATH 错误消息是由路径错误引起的，并最终传送到最初的发送端以告知路径出错；
- RESV 错误消息是由资源预留请求消息引起的，并传送到最终的接收端以告知预留出错消息，例如某个节点无法支持预约的资源。

4）拆链消息

拆链消息用于删除路径和资源预留状态，包括两种类型：PathTear 消息和 ResvTear 消息，节点（路由器）状态的删除可能会引起本节点相关预约状态的更新。

- **PathTear** 是针对 PATH 消息的，即删除从消息发送节点到所有消息接收者路径上的预约状态，PathTear 消息的路由与 PATH 消息严格一致。
- **ResvTear** 是针对 RESV 消息的，即删除从消息接收节点到所有消息发送者路径上的预约状态，ResvTear 消息的路由与 RESV 消息严格一致。

RSVP 消息由三部分构成：RSVP 消息头段、RSVP 对象段和 RSVP 对象内容。消息头段的结构如图 23－23 所示，其中“类型”（第三个域）用于标注上述 4 种报文类型的细分场景（如表 23－6 所示）。

4	4	8	16	16	8	8	32	7	1	24
版本号	标志	类型	校验和	长度	预订	发送 TTL	消息 ID	预订	MF	偏移

图 23－23　RSVP 消息头段的结构（初始行代表占用的位数）

表 23－6　PSVP 协议报文类型

1	路径
2	资源预订请求
3	路径错误
4	资源预订请求错误
5	路径断开
6	资源预订断开
7	资源预订请求确认

RSVP 对象段包含三部分，如图 23－24 所示。其中：

16	8	8
长度	分类号	C－类型

图 23－24　RSVP 对象段的结构（第一行代表占用的位数）

- **长度**：以字节为单位的对象总长度。

➢ **分类号**：表示对象类型。该分类号没有具体定义，只要能被 RSVP 识别即可。

➢ **C-类型**：可与分类号一起定义每个对象唯一的 ID。

RSVP 资源预约的流程大致是这样的：

（1）发送端首先向目的端发送 PATH 消息，用于描述发送端的数据格式、源地址、源端口号等流量与路径特征。

（2）目的端通过 PATH 消息了解到发送端的反转路径并决定哪些资源被预留。

（3）目的端根据上述信息反向发送包含资源预留参数的 RESV 消息给上游的路由器，以便它们建立预留状态并定期更新。

23.2 传输层协议

在我们日常的开发工作中，传输层协议是与应用软件联系最为紧密的协议层。一般来说，操作系统采用 socket 机制对网络层和传输层进行了很好的封装，我们无需过多关注底层细节，但因为传输层之上就是负责各种业务功能的应用层协议，应用层协议的某些传输特性取决于传输层，因此了解传输层协议是十分重要的。

在这里我们不准备对非常成熟并且资料翔实的 TCP 和 UDP 进行赘述，转而介绍一些较为少见的、近些年刚刚出现的传输层协议（SCTP、QUIC、UDT 等）。

传统的传输层协议一般是指 TCP 和 UDP，这两种协议各有优缺点。当前新出现的传输层协议一般是基于这两个协议做的优化和增强，特别是借鉴 UDP 的快速传输特性和 TCP 的拥塞控制特性衍生的新协议，这就是我们常说的 RUDP（Reliable UDP，可靠 UDP）。本节中的 SCTP、QUIC、UDT 等都是基于这种方式实现的。虽然这些协议基于 TCP 和 UDP，“看上去”应该属于传输层之上的应用层，但这些协议都是为应用服务的，都是具体的应用报文承载协议，因此我们将其都归类为传输层协议。还要强调一点，这里所说的应用层是指 TCP/IP 五层模型中的应用层，而不是 OSI 七层参考模型中的应用层。

除了上述几种传输层协议，目前也有许多特殊领域的传输层协议是基于 RUDP 的，例如流媒体领域的 RTP/RTCP，本质上也是在 UDP 的基础上增加了丢包反馈等可靠性机制，也可以看作 RUDP 的一种简单实现。还有一些领域则直接在 UDP 上嫁接了改进的定制化的 TCP 拥塞控制算法。正因为数据传输本身和拥塞控制本来就是相对解耦的两种技术，而近年来拥塞控制算法也层出不穷并得到了很大的增强，因此这样的“嫁接”也是可行的。

TCP 经典拥塞算法分为 4 个部分：慢启动、拥塞避免、拥塞处理和快速恢复，这 4 个部分都是为了控制发送窗口和发送速度而设计的，其实就是为了在当前网络条件下通过网络丢包来判断网络拥塞状态，从而确定比较合适的发送窗口。由于 TCP 拥塞控制已经广泛应用于 RUDP 中了，因此本节也会介绍一些控制算法。

RUDP 在拥塞控制中对于重传的要求包括定时重传、请求重传和 FEC（前向纠错）选择重传三种。

➢ **定时重传**：发送端在一个 RTO（Retransmission TimeOut，重传超时时间）周期后并未收到发送数据包的 ACK 确认消息，这可能是由于 ACK 报文丢失或链路过长导致 ACK 仍在途等原因造成的。但在定时重传要求下发送端会认为已经丢包并重传先前的数据包。

➢ **请求重传**：接收端在发送 ACK 确认消息的时候携带丢失报文的包号或者序列号信息，比如 UDT 协议中的 NAK（Negative Acknowledgement，否定确认）反馈就是采用这种方式。发送端在接收到这种 ACK 确认消息后根据包号或序列号选择性重传。

➢ **FEC 选择重传**：发送端发送报文时，会根据 FEC 的算法将报文分组，再通过 XOR（异或）的方式得到若干冗余包，并伴随着数据包一起发给接收端。接收端发现丢包时通过冗余包和分组内其他剩余数据包可以还原丢失包。但这不是万全之策，因为如果包丢得太多就无法还原了，这时要向发送端请求原始数据包。

下面我们就来介绍具体的传输协议。

23.2.1　SCTP

SCTP（Stream Control Transmission Protocol，流控制传输协议）是一种兼有 TCP 和 UDP 两种协议特征的新的传输层协议。SCTP 借鉴 UDP 的优点解决了 TCP 的某些局限，但其可靠传输和拥塞控制的机制基本上继承于 TCP，并做了改进和完善。SCTP 与 TCP 一样基于网络层协议，也使用五元组（源地址、源端口、目的地址、目的端口、协议号）来唯一标识一个连接（SCTP 中称为耦联），而且也支持 bind、listen、connect、accept、close 等操作，可以说 SCTP 就是 TCP 的加强版。下面我们列举一些 SCTP 独有的特性。

1. SCTP 的特性

1）多宿主（Mulit-Homing）机制

多宿主也叫作多路径传输，即通信的两端都可以绑定到多个 IP 地址上，只要有一对 IP 地址能够互通则 SCTP 的耦联（相当于 TCP 中的连接）就可以用。在多宿主机制下端口必须是唯一的，也就是说支持 SCTP 绑定多 IP 地址，但不支持绑定多端口，TCP 的单宿主机制与 SCTP 的多宿主机制如图 23－25 所示。

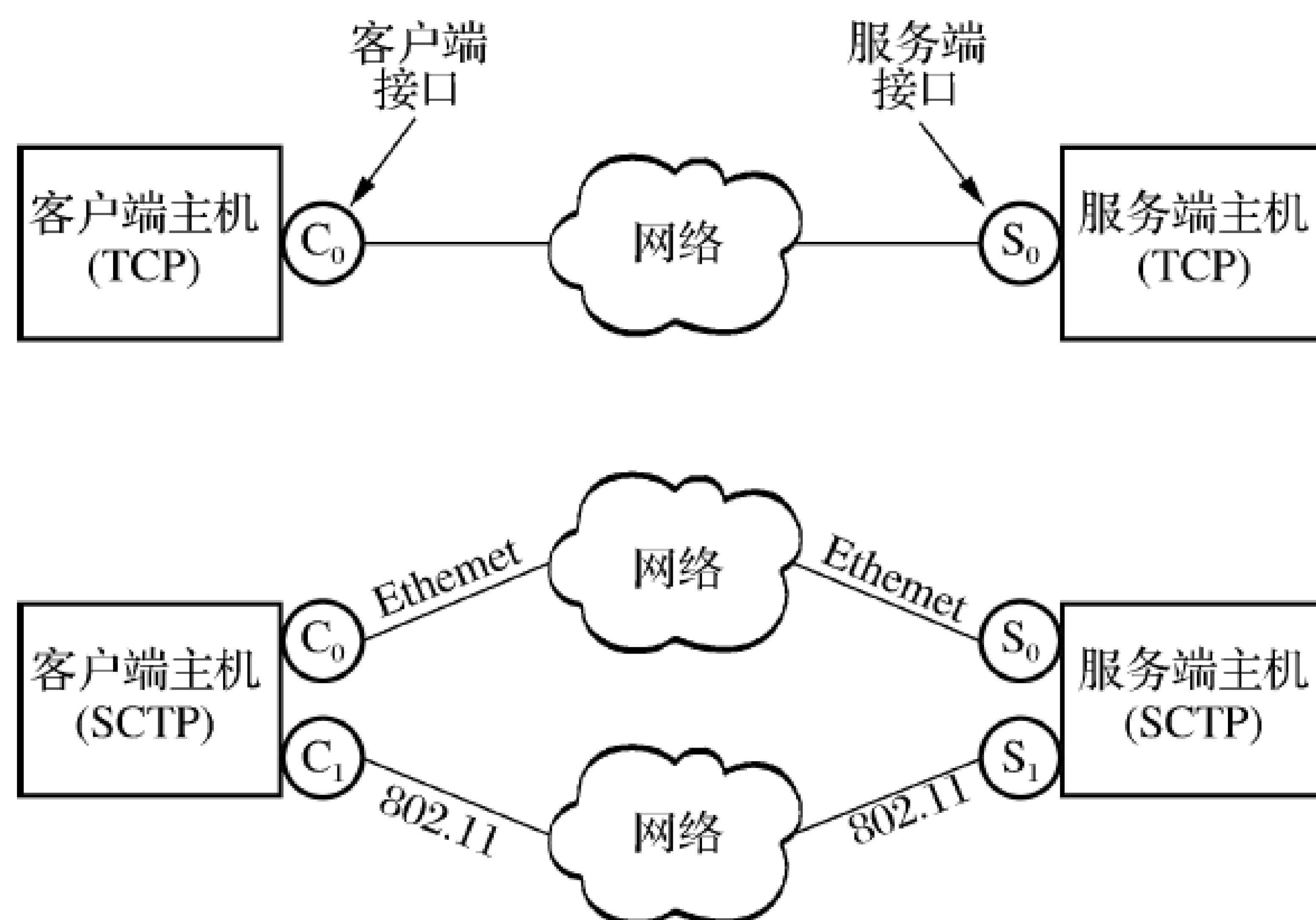


图 23－25　TCP 的单宿主机制与 SCTP 的多宿主机制

在多宿主机制中，若一条路径失效，SCTP 会通过另一条路径来发送数据，应用程序不必知道发生了故障和切换，也不必关注如何恢复。发生故障时，SCTP 首先切换通信对端的地址，如果切换后仍无效则切换本端地址。SCTP 使用内嵌的心跳保活机制来维持耦联中各条路径的可用性。

2）多流（Mulit-Streaming）机制

在 SCTP 的耦联中使用流来表示需要按顺序提交到高层协议的用户消息的序列，一个耦联中可以存在多条流，每一条流中的消息都会按顺序提交到上层，如图 23－26 所示。

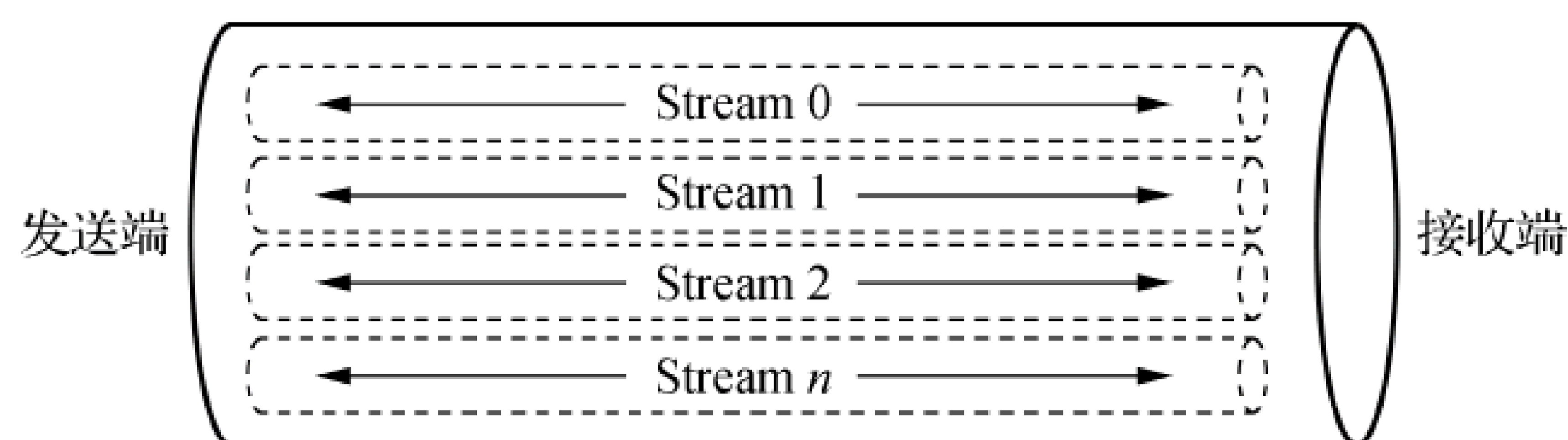


图 23－26　耦联中的多流机制

每条流都有唯一的流编号，这个编号是编码到 SCTP 的报文头中的。在耦联的多条流中，阻塞的流理论上是不会影响到其他流的。在同一条流中，SCTP 支持有序传输和无序传输两种方式。在调用 SCTP 的 sendmsg 方法时要指定在哪一条流上用哪种方式来进行传输。

- 有序传输以顺序优先，在遇到丢包等情况的时候可能会在接收端被阻塞（例如等待丢失的包重传）。有序传输是 SCTP 流传输的默认方式。
- 无序传输以效率优先，不会在接收端被阻塞，即使遇到丢包等情况也不需要傻等重传丢失的包。
- 无序传输多用于面向消息的协议，这是因为大部分消息本身都是独立的，与消息的次序没有关系。
- 在 SCTP 流中可以配置无序传输方式，也可以在一个耦联中选择性地对部分流配置无序传输方式。

3）初始化保护机制

在 SCTP 中，初始化保护机制主要是指连接建立时的保护。与 TCP 不同，SCTP 建立连接需要四次握手，其过程如图 23－27 所示。

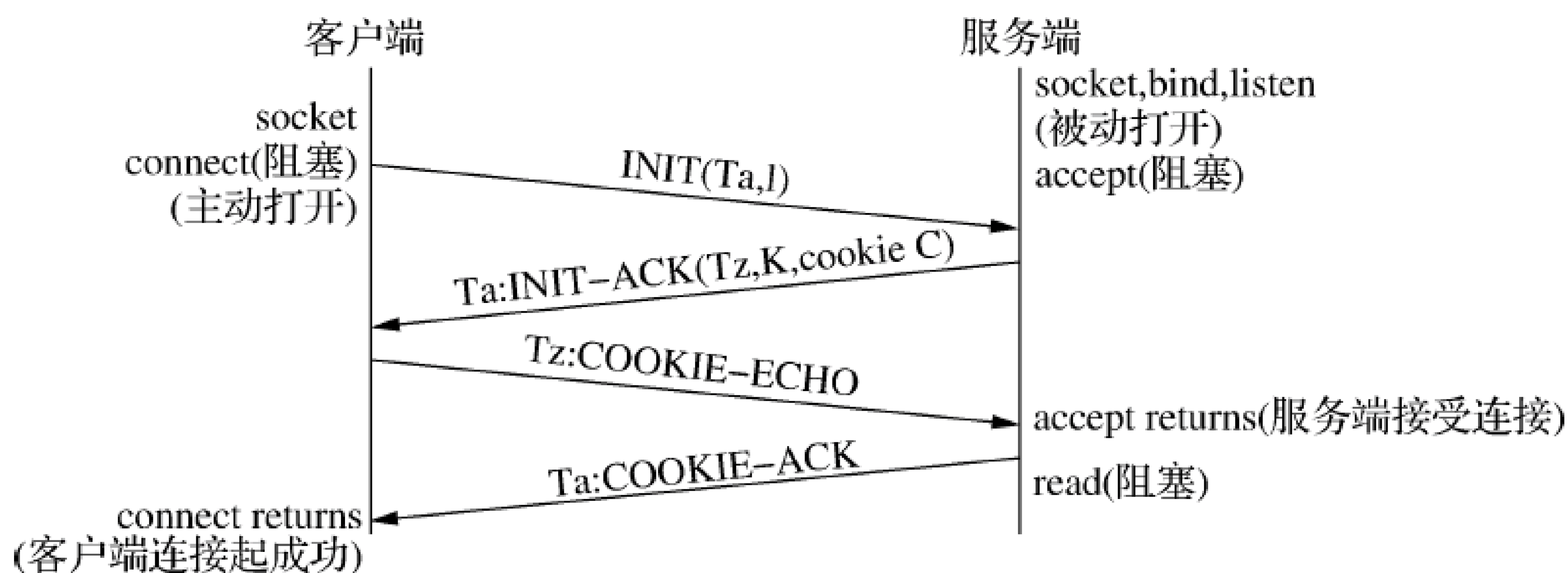


图 23－27　SCTP 的四次握手过程（初始化）

Ta、Tz——建立连接时双方使用的验证标志

SCTP 也分为客户端和服务端,这一点与 TCP 是一样的,并且握手协商都是由客户端发起的。

（1）客户端首先向服务端发起 INIT 消息。

（2）服务端回复 INIT-ACK 消息作为应答,该报文中要携带服务端生成的 Cookie 信息作为标识本次连接的唯一上下文。

（3）客户端再次向服务端发送 COOKIE-ECHO 报文响应,报文中包括步骤(2)中的 Cookie 信息,以便服务端进行验证。

（4）服务端对上述 Cookie 进行验证,当 Cookie 正确时才分配连接通信的相关资源,并向客户端回复 COOKIE-ACK 消息作为应答。

从上述流程可以看出,在 SCTP 的四次握手协商中,前面两次客户端与服务端都不需要保存连接状态,也不需要分配资源,以此来化解 SYN Flood 攻击的风险。而在 TCP 的三次握手过程(如图 23－28 所示)中,由于在服务端接收到 SYN 请求后就开始分配资源保留状态(SYN_RCVD 状态),因此面对 SYN Flood 攻击时只有不断地消耗资源,最终导致资源耗尽。

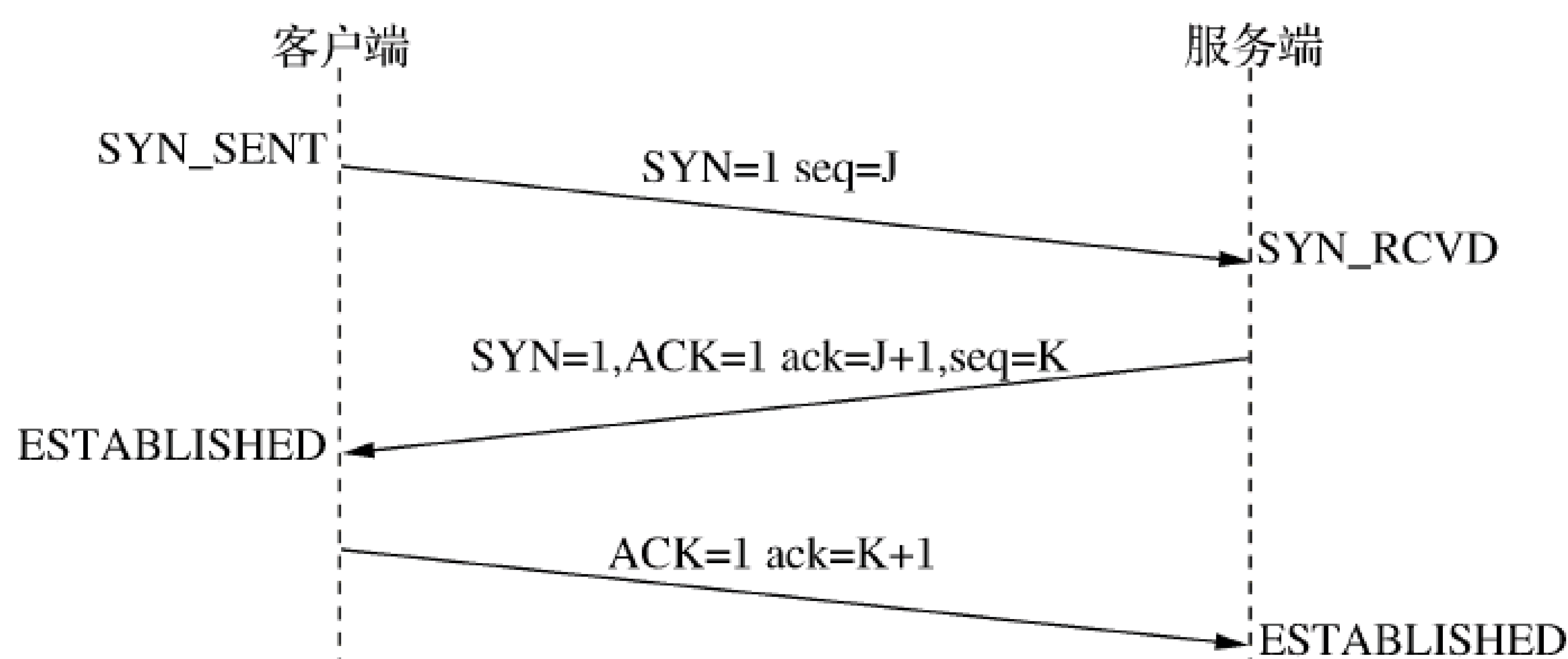


图 23－28　TCP 的三次握手过程(初始化)

另外,因为要使用四次握手机制,必然会带来一定的数据延迟问题。SCTP 允许把数据包含到 COOKIE-ECHO 和 COOKIE-ACK 报文中进行合并发送,以提前开始数据传输来减少延迟,因此反而比 TCP 更早地“进入状态”。

4）消息分帧机制

TCP 是面向字节的,SCTP 则是以数据块/消息为传输单位的,以此来保护消息边界。

- 在 SCTP 的发送过程中,应用进程调用 sendmsg 接口时以该函数发送的数据作为一个整体的消息数据块,并且放在 SCTP 报文的 Data Chunk 数据结构中发送。这一点就比较类似 UDP 的发送方式了,边界非常明确。
- 如果一条消息过长(例如超过了链路中的 MTU),SCTP 发送时也会将消息数据拆分成多个片段并封装在多个 Data Chunk 数据结构中,再通过多个 SCTP 包发送给接收端。
- 在 SCTP 的接收过程中,接收端进程调用 recvmsg 接口时收到的都是一条完整的消息数据,一般来讲其边界与发送时 sendmsg 接口封装的一致,这是因为接收端也是以 Data Chunk 为单位来接收和解析数据的。
- 发送多条短消息时可以被 SCTP 发送端合并成一条 SCTP 消息(包含多个 Data Chunk

结构）发送以提高发送效率，接收端接收时也同样可以看到多个一一对应的 Data Chunk 数据结构。

可以看出，在 SCTP 的传输过程中，Data Chunk 是数据分界的"容器"。

5）平滑关闭机制

与 TCP 不同，SCTP 采用了三次握手关闭耦联的机制，我们称之为"平滑关闭"，这是与 TCP 相比而言的"平滑"。在 TCP 中采用了四次握手机制来结束连接会话，每一方若想结束会话就要进行"FIN-ACK"协商交互，也就是说在 TCP 双工的基础上，连接可以半关闭（一次 FIN-ACK 握手），也可以全关闭（两次 FIN-ACK 握手）。

SCTP 简化了关闭握手协商，不支持半连接或者半关闭状态。因为在实际使用过程中，半关闭其实没有太大的用处，反而徒增了协商的复杂性。类似于 TCP 的初始化握手，SCTP 关闭耦联采用了三次握手，如图 23－29 所示。

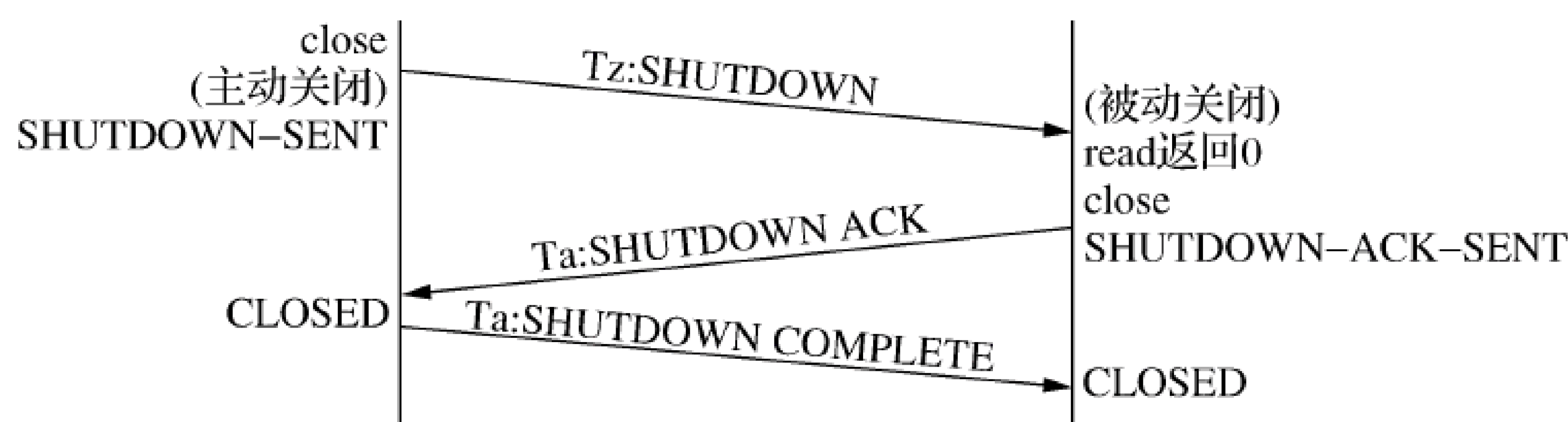


图 23－29　SCTP 的三次握手过程（关闭耦联）

6）选择性确认机制

与 TCP 不同，SCTP 的确认号（SACK）是用于确认丢失数据包的。TCP 中确认序号返回的是发送端已成功收到数据字节序号（不包含确认序号所指的字节），而 SCTP 反馈给发送端的是丢失的并且要求重传的消息序号。这样就大大减少了服务端需要发送的回复数量，做到了"有事启奏，无事退朝"般的高效。

7）心跳保活机制

SCTP 协议本身支持心跳保活机制（Heartbeat）来监控耦联或路径（Path）的可用性，不同的路径都可以由心跳或者数据的传输/确认来监控其状态（是否可用，是否还保持连接状态）。这种心跳保活机制是上层的应用进程感觉不到的。而 TCP 则需要上层应用进程自主地发送心跳以保活连接。

从以上描述中可以看出，SCTP 既与 TCP 相似，也有对于 TCP 的改进之处，特别是其报文格式与 TCP 有了很大的不同。最主要的改进是引进了适应数据块传输的 Data Chunk 结构，这大大改善了数据分界特性，而这一点在 TCP 中往往是通过进程在应用层协议中实现的。

2. SCTP 报文结构

SCTP 报文也分为头域与消息体两大部分，其中消息体以 Data Chunk 作为分界单位，每个 Data Chunk 包含了一个消息数据块。一个 SCTP 可以包含多个 Data Chunk，只要总的大小不超过链路中的 MTU 就行（INIT、INIT-ACK、SHUTDOWN 报文除外，对这种类型的消息，

SCTP 报文只能承载一个 Data Chunk）。Data Chunk 既可以承载控制信息也可以承载用户进程的数据信息。SCTP 报文结构如图 23－30 所示。

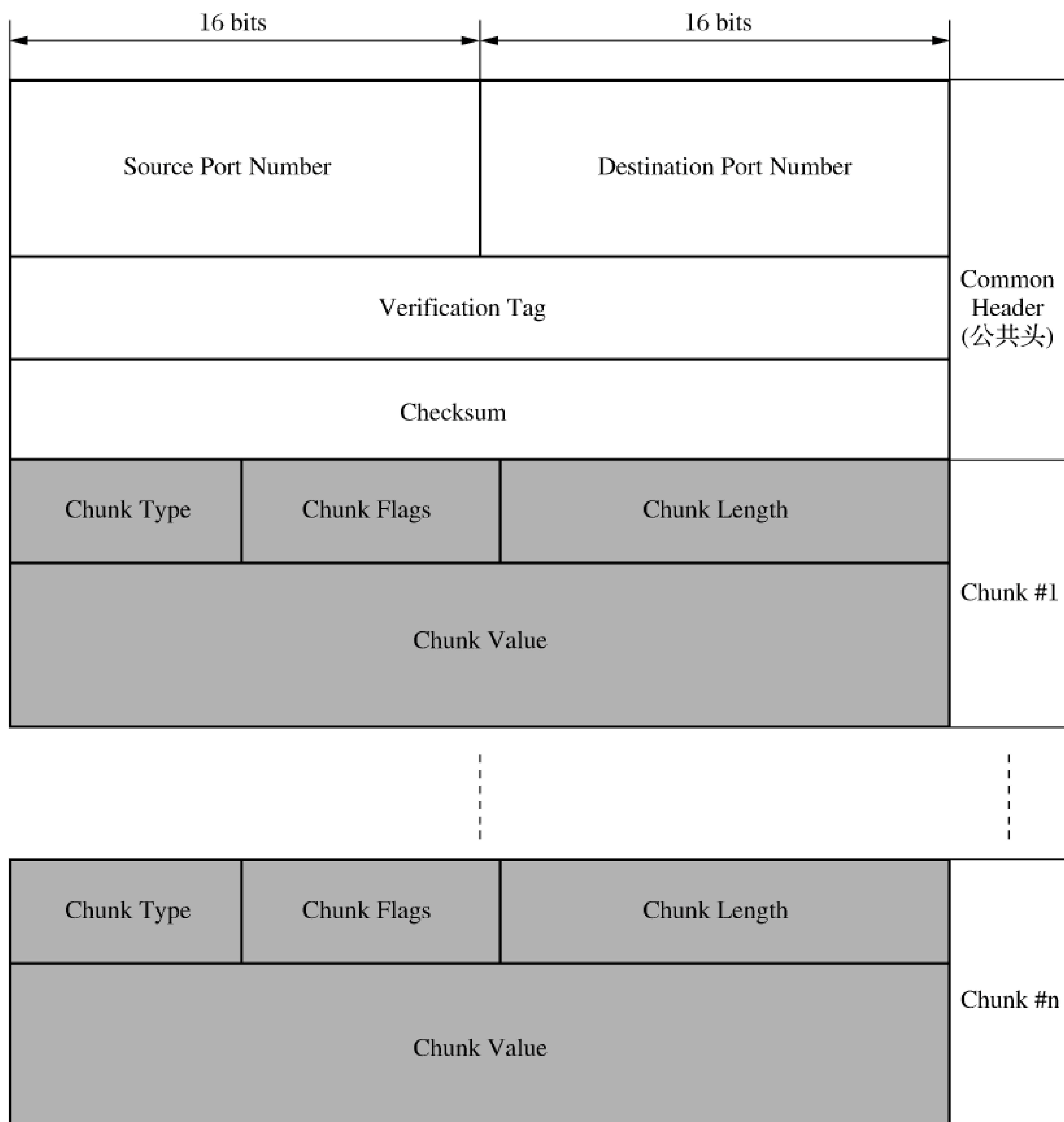


图 23－30　SCTP 报文结构

SCTP 报文头占 12 个字节，其中包含了源端口号和目的端口号，这一点也是四层协议的特征（传输层是要解决端口问题的）。同时，报文头中还包括了验证标签和校验码信息。上述几个属性统称为 SCTP 的公共报文头。

- **验证标签（Verification Tag）**：建立耦联时，发起端为该耦联生成一个 32 位的随机数字标识（Tag）。耦联建立过程中，双方会交换这个 Tag。在数据传递时，发送端必须在公共报文头中带上对端的这个 Tag 以备校验，从而防范中间人攻击。
- **校验码（Checksum）**：用于数据完整性校验，由数据发送端产生，接收端负责验证。

报文头后面紧接着报文体，报文体是由一个或多个 Data Chunk 组成的。Data Chunk 有自己的类型，表 23－7 描述了数据和控制两种报文的 Data Chunk 类型。

表 23－7　Data Chunk 的类型值

Data Chunk 标识值	Data Chunk 类型	Data Chunk 标识值	Data Chunk 类型
0	DATA	12	ECNE
1	INIT	13	CWR
2	INIT-ACK	14	SHUTDOWN COMPLETE
3	SACK	15～62	被 IETF 保留
4	HEARTBEAT	63	块扩展
5	HEARTBEAT ACK	64～126	被 IETF 保留
6	ABORT	127	块扩展
7	SHUTDOWN	128～190	被 IETF 保留
8	SHUTDOWN ACK	191	块扩展
9	ERROR	192～254	被 IETF 保留
10	COOKIE ECHO	255	块扩展
11	COOKIE ACK		

➢ **块类型（Chunk Type）**：是在 Data Chunk 的第一个字节中体现的，这是个单字节的属性，表示了块值（Chunk Value）的类型。对于接收端不能识别的块类型（例如用户自己扩展的数据类型），Chunk Type 的高两位标识了对这些块的处理操作，这些操作包括：
  - 0x00：停止处理并丢弃当前报文，不再处理该报文中其他的 Data Chunk。
  - 0x01：停止处理并丢弃当前报文，也不再处理该报文中其他的 Data Chunk，并且需要在 ERROR 或 INIT-ACK 报文中向发送端返回这个不能识别的块类型值。
  - 0x10：跳过该 Data Chunk 继续处理其他的块。
  - 0x11：跳过该 Data Chunk 继续处理其他的块，并且需要在 ERROR 或 INIT-ACK 报文中向发送端返回这个不能识别的块类型值。

➢ **块标志位（Chunk Flags）**：用法由块类型决定。

➢ **块长度（Chunk Length）**：块类型、块标志位、块长度和块值几部分的总长度，使用二进制格式表示。

➢ **块值（Chunk Value）**：包含的数据内容。

23.2.2　QUIC

作为传输层协议的 QUIC（Quick UDP Internet Connection，快速 UDP 互联网连接）协议有两种形态，一种是由 Google 研发制定的 gQUIC（Google QUIC）协议，另一种则是由 IETF 提出的 iQUIC（IETF QUIC）协议，后者是对前者的改进和增强。但无论是哪种，本质上 QUIC 协议都是使用 UDP 进行多路并发传输的协议，其宗旨也都是加速 HTTP 通信并使其更安全，以最

终在 Web 上代替 TCP 和 TLS 协议。因此 QUIC 在功能上等价于 TCP + TLS + HTTP2.0。与 TCP 相比，QUIC 具有许多新特性。

1. QUIC 的特性

1）连接建立延时低

建立 Web 连接时，TCP 方式下要首先建立 TCP 连接（三次握手），之后要建立 TLS 连接（三次握手），最后才是 HTTP 协议报文的交互；而在 QUIC 方式下，只需要建立 QUIC 连接（三次握手）即可在其上运行 HTTP 协商交互，如图 23－31 所示。这是因为 QUIC 用它自己的框架格式代替了 TLS 协议的记录层，同时保留了与原来相同的 TLS 握手消息。因此一次 QUIC 协商相当于 TCP + TLS 的两次协商。

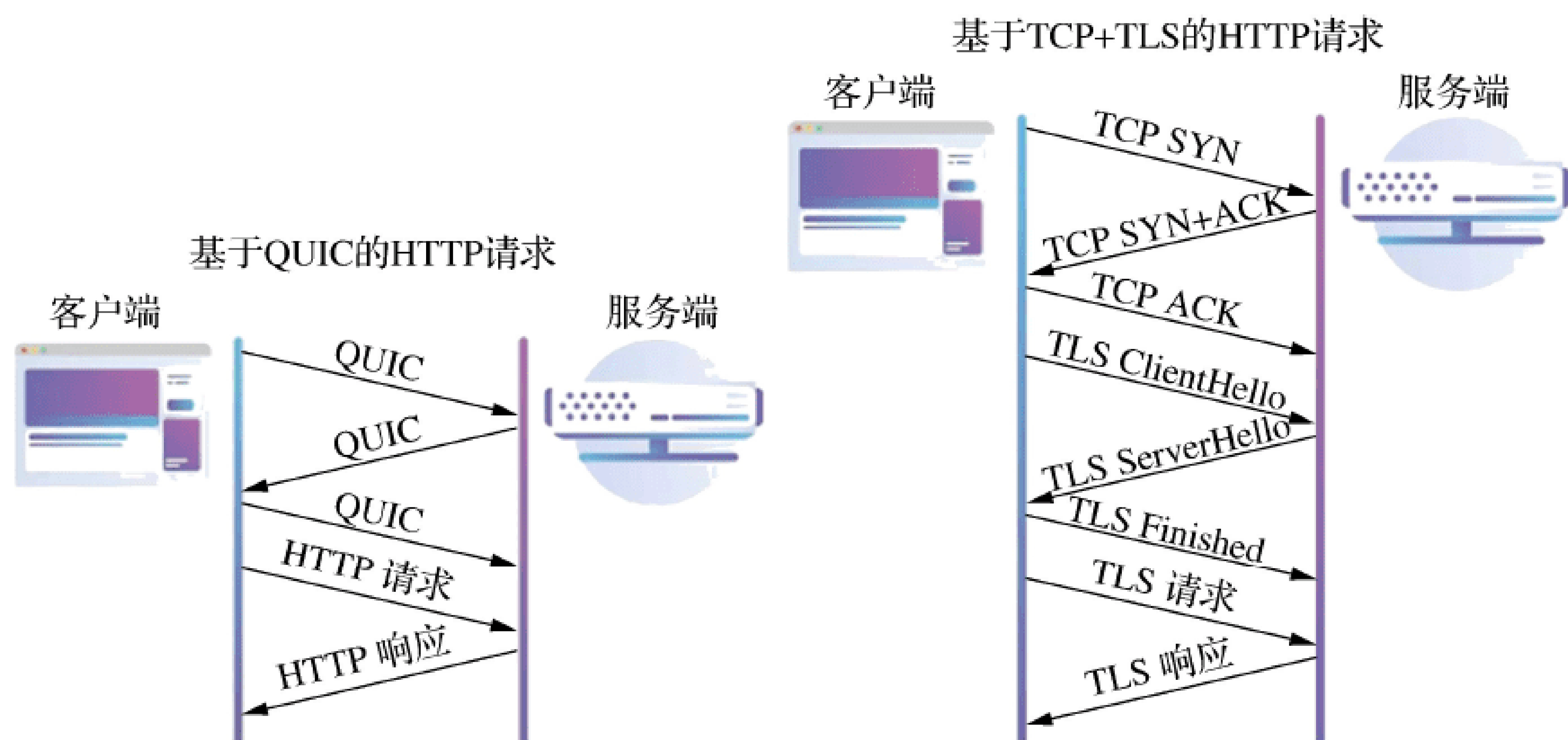


图 23－31　QUIC 与 TCP + TLS 协议上的 HTTP 交互过程对比

2）改进的拥塞控制算法

TCP 采用传统的慢启动 + 拥塞避免 + 快速重传 + 快速恢复的拥塞控制机制，QUIC 协议则默认使用 TCP 的 Cubic 拥塞控制算法，同时还支持 CubicBytes、Reno、RenoBytes、BBR 和 PCC 等拥塞控制算法。

在 TCP 中，慢启动的规则是这样的：

- 当 cwnd < ssthresh 时，使用慢启动算法，这一阶段 cwnd 是翻倍增长的（1、2、4、…）；
- 当 cwnd > ssthresh 时，停止使用慢启动算法而改用拥塞避免算法，这一阶段 cwnd 是线性增长的（N、N + 1、N + 2、…）；
- 当 cwnd = ssthresh 时，既可使用慢启动算法，也可使用拥塞避免算法。

在这里，cwnd 代表拥塞窗口的大小，ssthresh 表示慢启动阈值。慢启动和拥塞避免是 TCP 拥塞控制的两个阶段。在 TCP 中发送端考虑到了接收端的接收能力，因此让己端的发送窗口小于等于拥塞窗口的大小（cwnd）。慢启动的核心思想就是一开始先发送少量的数据来探测下网络的拥塞程度，再由小及大地增加拥塞窗口的大小，直到拥塞窗口的大小赶上慢启动阈值（ssthresh），赶上和超过以后，TCP 发送端就不能再这么疯狂恣意地发送了，这时应采用渐进式的发送策略（拥塞避免阶段），使用拥塞避免算法线性地增大拥塞窗口。

无论在慢启动阶段还是在拥塞避免阶段，只要发送端判断网络出现拥塞（其根据就是没有收到 ACK 确认消息），就要把慢启动阈值 ssthresh 设置为出现拥塞时的发送端窗口值的一半（但不能小于 2），然后把拥塞窗口的大小 cwnd 重新设置为 1，再开始执行慢启动算法。

而在 QUIC 协议中，拥塞控制机制具有以下新特性：

- **支持可插拔的拥塞控制算法**：可以在不停机的情况下灵活地启动、变更和停止拥塞控制算法，应用进程可在不需要内核支持的情况下启用不同的控制算法。
- **精细的拥塞控制算法实施粒度**：支持同一进程在不同 QUIC 连接情况下采用不同的拥塞控制算法。
- **以 Packet 为传输单位**：对比 TCP 中以字节为单位的流传输方式，QUIC 是以 Packet 为单位传输数据的，每个 Packet 都有自己的 Packet Number，且 Packet 的大小不会超过 MTU，这更像 UDP 的传输方式。
- **单调递增的 Packet Number**：类似于 TCP 报文中的 Sequence Number 来保证消息的有序性。但是 Packet Number 严格递增，即使某个包丢失了，重传的包的序号也是大于当前 Packet Number 的，可谓"既往不咎"。Packet Number 的单调递增解决了 TCP 重传的歧义性。
- **采用 Stream Offset 保证数据的顺序性和有序性**：在 QUIC 中 Stream 相当于一条 HTTP 请求，对于 Stream 而言，传输的单位是 Packet。QUIC 采用 Stream Offset 来保证数据的顺序性和有序性，因为 Packet Number 是"不回头"的，如果发生重传，发送端就不会知道到底要重发哪一个 Packet，因此只能用 Stream Offset 来代表流数据的偏移以保证数据的顺序性。
- **更加精确的时间统计**：在 TCP 方式下，ACK 回送时只是将 SYN 中的时间戳传递进去，并没有计算接收端处理 SYN 和发送 ACK 之间的时间间隔，因此存在"ACK Delay"的时间空隙。QUIC 协议没有忽略这个间隙，因此在 QUIC 中 RTT（Round-Trip Time，往返时延）也将这一部分时间算了进去，如图 23－32 所示。

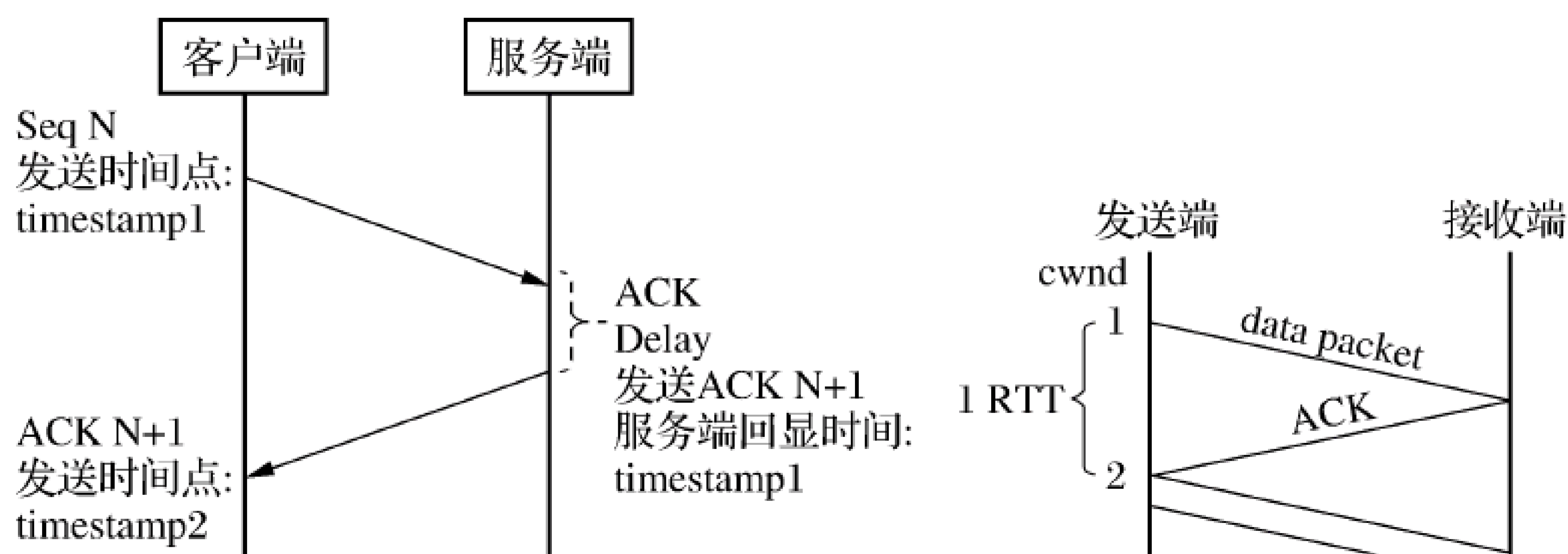


图 23－32　TCP 中的 ACK Delay 与 QUIC 中的 RTT

3）基于 Stream 和 Connection 级别的流量控制机制

Connection 可以看作一条 TCP 连接，因此可以认为 Connection 是四层的，Stream 是五层及以上的。QUIC 要支持多路复用，其实就是基于一条 Connection 上面多路的 Stream 而言

的。QUIC 的流量控制既可以对单条 Stream 进行流量控制，也可以对所有的 Stream 进行流量控制。在 Connection 中，有一个专门的流（Stream ID = 1）用于执行握手协商。

4）没有队头阻塞的多路复用机制

在 TCP 中，当有数据丢失时，哪怕只丢了一个字节，也要等待重传，因而导致后面的数据被阻塞住。TLS 协议也存在同样的问题，虽然是以 Record 为处理单位，但加密往往跨越多个 Record，即使其中只丢失了一个字节也无法正常解密。上述这种情况被称为“队头阻塞”。

但在 QUIC 协议中传输的基本单元是 Packet，整个认证和加密也都是基于 Packet 的，且不会跨越多个 Packet 加密，因此即使丢包也只需要重传这一个问题 Packet，其他的包不依赖于这个 Packet 也可以正常解密。而且一个 Connection 上的 Stream 彼此之间是独立的，不存在依赖关系，丢了再重传问题 Packet 就行了，后续的 Packet 该怎么接收还怎么接收，因此也不存在队头阻塞问题。此外由于 QUIC 支持前向纠错（FEC）机制，因此即使有少量的丢包，在接收端完全可以基于这种机制计算出丢失的数据，不需要重传，减少了阻塞的发生。

5）数据加密机制

TCP 报文头部并没有任何加密或认证措施，这样做虽然保证了解析效率，但也确实存在安全隐患，因此监听到的都是明文，很容易实施中间人攻击。

QUIC 协议内置了 TLS 协议栈，以此实现了传输层加密。对于绝大部分报文，头部需要经过认证（QUIC 报文头带有认证字段），报文体整体需要经过加密。其中加密采用了初始密钥和会话密钥的两级密钥协商机制：

- 初次连接时不加密，但开始协商初始密钥；
- 待初始密钥协商完毕，马上协商会话密钥；
- 通信过程中接收端可以根据需要更新密钥，更新时会采用新旧两种密钥解密，解密成功的密钥才会被保留，否则即使是新密钥也不会更新（数据可用性优先）。

6）连接迁移

与 TCP 不同，QUIC 协议采用一个由客户端随机产生的 64 位随机数作为连接 ID（UUID），而不再采用四元组（源 IP 地址、源端口号、目的 IP 地址、目的端口号）这种第三或第四层协议的特征标识来识别一条连接。因此其中任何一个元素发生变化时（例如从 Wi-Fi 切换到 4G 或者 NAT 穿透时端口动态变化），这条连接依然维持有效，能够保持业务逻辑不中断，对于应用程序是无感的，不需要重新建立连接。

7）前向纠错

QUIC 协议中每个数据包除了包含它本身的内容之外，还包含了部分其他数据包的数据，因此少量的丢包可以通过其他包的冗余数据直接计算出来而无需重传，这种机制也被称为前向纠错（Forward Error Correction，FEC）。目前默认每发送 10 个数据包就包含一个冗余包，其冗余数据可以重新构建一个丢失的数据包。

2．QUIC 报文结构

QUIC 报文分为报文头（也被称为公共包头）和报文体两部分，报文头结构如图 23－33

所示，具有认证字段；报文体全部加密，保证了信息的私密性。

QUIC 报文分为普通报文和特殊报文两种形式，而前者又细分为帧报文和 FEC 报文；后者则细分为版本协商报文（Version Negotiation Packet）及公共重置报文（Public Reset Packet）。



图 23－33　QUIC 报文头结构（单位：位）

➢ **Flags**：用于说明 QUIC 报文的性质和其他字段的大小，具体用法如下：
- 0x01（PUBLIC_FLAG_VERSION）表示版本协商报文；
- 0x02（PUBLIC_FLAG_RESET）表示公共重置报文；
- 0x04 表示报文头包含了 32 字节的 Diversification Nonce；
- 0x08 表示报文头中包含了 64 位的 Connection ID，该标志位在所有的报文中都应该被设置；
- 0x30 表示字段占用 6 个字节；
- 0x20 表示字段占用 4 个字节；
- 0x10 表示字段占用 2 个字节；
- 0x00 表示字段占用 1 个字节；
- 0x40 为多路径使用保留；
- 0x80 当前未使用，且必须被设置为 0。

➢ **Connection ID**：这是一个由客户端随机选择的最大长度为 64 位的无符号整数，其长度也可以为 0、8 或 32 位。

➢ **Version**：该值为可选字段，代表了 QUIC 协议的版本号，长度为 32 位。Version 一般出现在协商的第一个包中，用于确保服务端可以保持与客户端规定的版本的一致性。

➢ **Diversification Nonce**：由 4 字节的时间戳、8 字节的服务器轨道和 20 字节的随机数组成的 32 字节的随机数。

➢ **Packet Number**：其长度由 Flags 定义，普通报文由发送者分配包号，特殊报文则由接收者分配包号。

QUIC 报文体（Packet）由帧（Frame）组成，报文体可以包含多个帧，而帧具体分为：流帧（STREAM）、ACK 帧、停止等待帧（STOP_WAITING）、窗口更新帧（WINDOW_UPDATE）、拥塞信息帧（BLOCKED）、拥塞反馈帧（CONGESTION_FEEDBACK）、填充帧（PADDING）、重置流帧（RST_STREAM）、连接关闭帧（CONNECTION_CLOSE）和 Ping 帧。无论一个 Packet 由多少帧构成，帧都不能跨越 Packet。

上述类型的帧又可分为特殊帧和常规帧两类。特殊帧在 Type 字段中同时编码了帧类

型和相应的标志位;而常规帧只是编码了帧类型。除了最常见的流帧,ACK 帧和拥塞反馈帧都是特殊帧,其余的帧则都是常规帧。

- **流帧**:用于承载多路复用流的应用数据,即隐式创建流和发送流数据。
- **ACK 帧**:用于通知发送端有哪些帧已经被接收端接收或者哪些帧没有被接收端收到。
- **停止等待帧**:用于通知对端不再等待小于指定包序号(Packet Number)的包,包序号的长度可以是 1、2、4、6 字节。
- **窗口更新帧**:用于通知对端本端的流量控制的接收窗口的扩容增长情况。
- **拥塞信息帧**:用来通知接收端本端有数据需要发送,并且已经准备好发送数据了,但被流量控制所阻塞。
- **拥塞反馈帧**:这是个实验性质的帧,当前没有被使用,其目的是在标准的 ACK 帧范围之外提供额外的拥塞反馈信息。
- **填充帧**:填充帧包含 0x00 的字节,用于填充 QUIC 包的末尾。当遇到了填充帧,该包的剩余部分都是填充帧。
- **重置流帧**:用于关闭一个流。当该帧由流创建者发送时,表示创建者希望取消这个流;当该帧由流接收端发送时,表示接收端发生了一个错误,或不希望接受这个流而应关闭。
- **连接关闭帧**:用于通知连接被关闭。如果还有流在传输,所有这些流都被隐式关闭。
- **Ping 帧**:用于验证对端是否还存活。接收端需要返回这个包的确认包。当流被打开时,需要用 Ping 帧来保证连接活跃。Ping 帧上没有负载信息。

上述每种类型的帧结构都是不一样的,例如用于创建流和发送数据的流帧需要带有与流相关的属性(如 Stream ID、Offset 等,如图 23-34 所示),而停止等待帧则不需要描述流信息,只需要描述与 Packet Number 相关的信息即可。

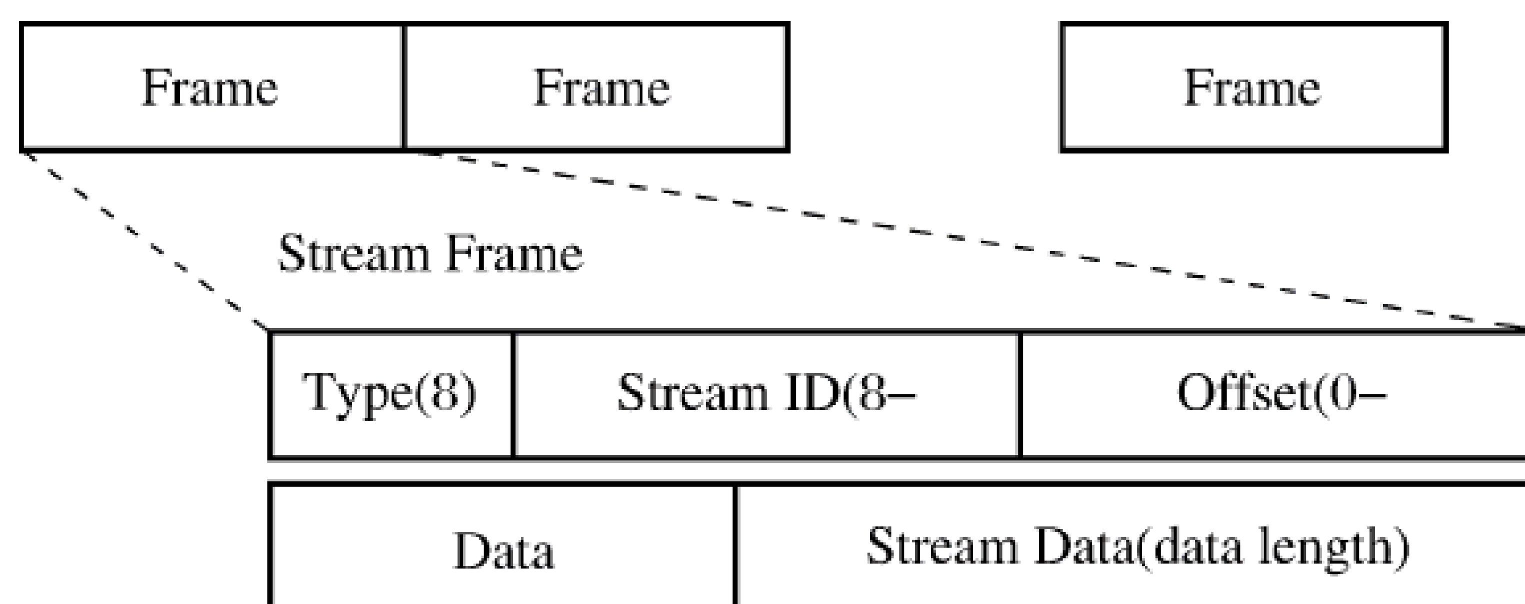


图 23-34　QUIC 流帧的结构(单位:位)

最后需要注意的是,QUIC 报文的大小需要满足 MTU 的上限以避免被分片,因此报文的大小在 IPv4 下最大为 1 370 字节,在 IPv6 下最大为 1 350 字节。

3. QUIC 的握手协商流程

QUIC 协议相当于 TCP + TLS + HTTP2.0,而其握手的过程则是 TLS 过程的体现。但这里要明确的是,QUIC 的连接建立与握手协商是两个层面的事情,QUIC 要先建立连接,在此

连接基础之上再进行握手协商。

QUIC 在握手过程中使用了 Diffie-Hellman 算法协商初始密钥。初始密钥依赖于服务端事先存储的一组配置参数,且这组参数会周期性地刷新。初始密钥协商成功后,服务端会再提供一个临时随机数,通信双方根据这个随机数再生成会话密钥。在整个会话过程中,由于会话密钥也依赖于初始密钥,因此会话密钥也会周期性地刷新。

QUIC 的握手过程比较复杂,可以根据客户端是否已知服务端的全部配置参数来区分对待握手流程,分别如图 23－35 和 23－36 所示。

1）客户端已缓存服务端的全部配置参数

（1）客户端向服务端发送 Complete Client Hello(CHLO)消息,该消息中包括了客户端选择的公开数。这一阶段客户端根据服务端的全部配置参数和自己所选的公开数可以计算出初始密钥。

（2）服务端收到 Complete Client Hello(CHLO)消息后,如果不同意就回复 Rejection(REJ)消息,该消息包括了部分服务端的配置参数。

（3）服务端如果同意,则根据客户端选择的公开数计算出初始密钥。之后向客户端回复 Server Hello(SHLO)消息,该消息采用初始密钥加密,其中也包含了服务端选择的一个临时公开数。

（4）客户端如果收到服务端的 REJ 消息,则提取并存储服务端的配置参数。

（5）客户端如果收到服务端的 SHLO 消息,则尝试基于初始密码进行解密并提取出临时公开数。

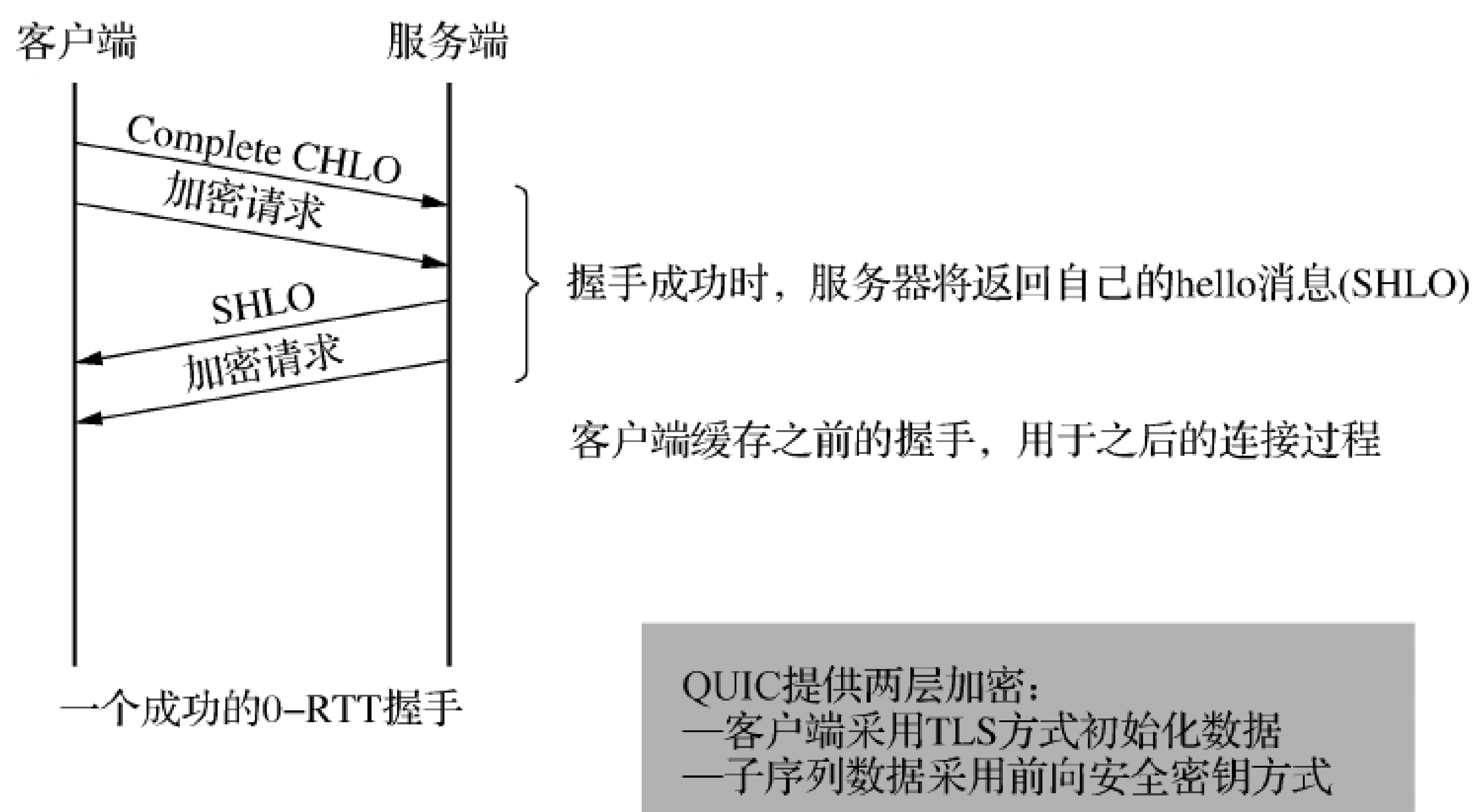


图 23－35　已缓存配置参数时的握手流程

（6）客户端与服务端各自根据临时公开数和初始密钥并基于 SHA-256 算法推导出会话密钥。

（7）双方刷新密钥为会话密钥(此时初始密钥可以丢弃),后续的通信过程使用会话密钥。至此握手过程完毕。

2）客户端未缓存服务端的全部配置参数或只缓存了一部分

(1) 客户端要向服务端发送 Inchoate Client Hello(CHLO)消息以获取服务端的全部配置参数。

(2) 服务端收到 CHLO 消息后回复 Rejection(REJ)消息,其中包含了服务端的配置参数。

(3) 客户端收到 REJ 消息后提取并存储服务端的配置参数。

(4) 接下来的流程与客户端已缓存服务端的全部配置参数时的流程一致。

可以看出,这种情况下首先需要获取和补齐服务端的全部参数,之后的流程就与已知全部参数的情况一致了。

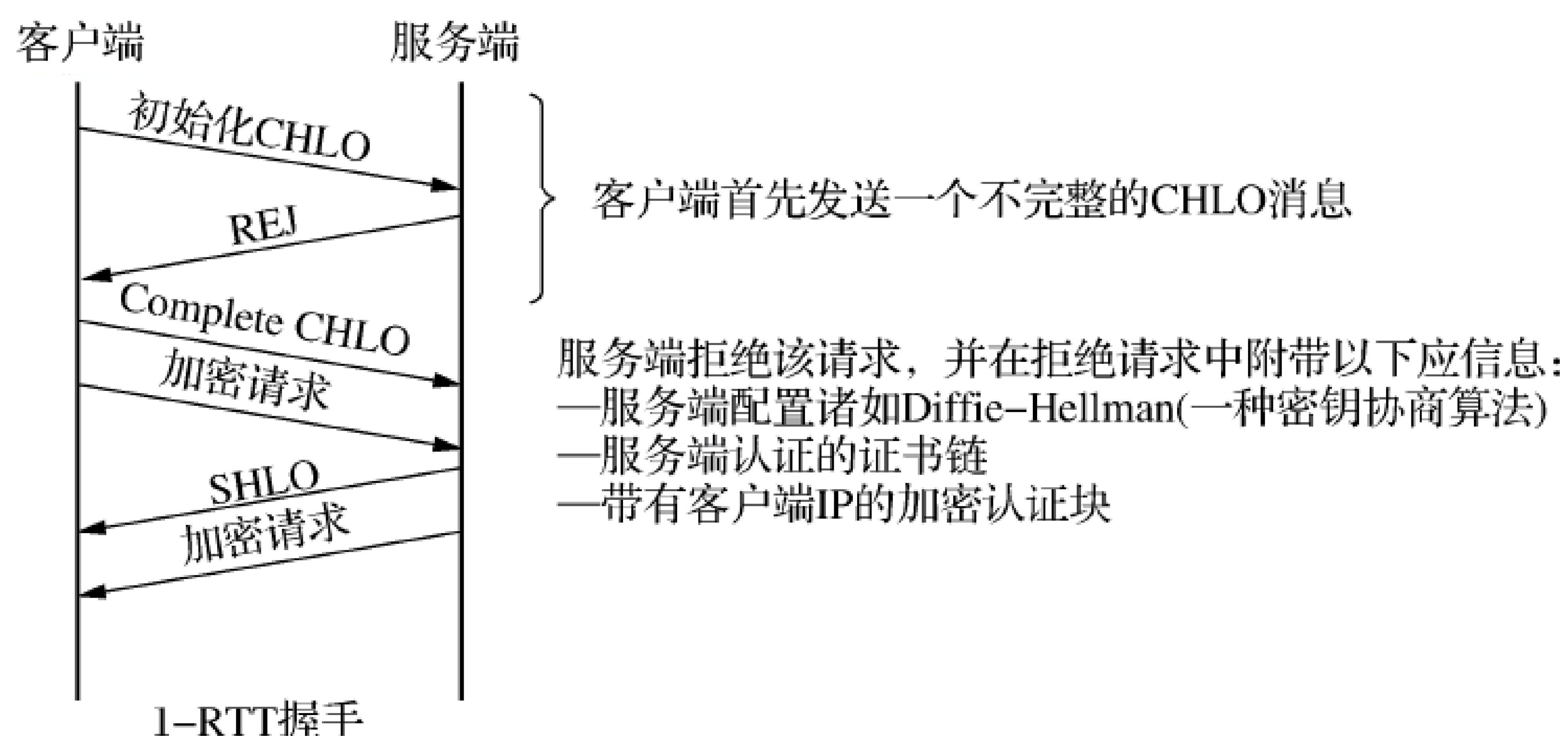


图 23-36　未缓存配置参数时的握手流程

23.2.3　UDT

UDT 是一种基于 UDP 的互联网传输层协议(UDP-based Data Transfer Protocol),其产生的背景是高带宽和长距离网络(例如互联网)上 TCP 性能不佳、带宽利用率低,因此需要一种在高 BDP(带宽时延乘积,即每秒比特与来回通信延迟的乘积,是网络性能指标之一)网络下具有良好传输性能和高带宽利用率的传输层协议。高 BDP 网络(High-Bandwidth-Delay-Product Network)代表了高带宽、高延时的网络,比如互联网,这种网络也被称为**长肥网络**(**Long Fat Network,LFN**)。

现有的 TCP 拥塞控制机制在高 BDP 网络下的表现并不理想:

- TCP 拥塞控制机制在 BDP 网络下拥塞窗口小并且增长缓慢。
- TCP 的 AIMD(Additive Increase Multiplicative Decrease,加性增大乘性减小)拥塞控制算法对于拥塞窗口的下降比较过激,却不能较快地恢复到窗口的高位值以恢复带宽。
- TCP 拥塞控制算法在高 BDP 网络下 RTT 公平性较差,严重影响了拥塞窗口的增长。

在这样的背景下,一种基于 UDP 的、速纳和改进了 TCP 拥塞控制且开源和双工的传输层协议被提出了,这就是 UDT。UDT 具有以下明显的特点和优势:

- **新的拥塞控制算法**:在新算法中,慢启动阶段可快速扩张拥塞窗口以抢占带宽,接近饱和状态时逐渐降低窗口增长速度并趋于稳定,这是一种基于增长速率的拥塞控制



算法。

➢ **可扩展的拥塞控制算法框架**:可以对不同的拥塞控制算法做适配,应用进程可以自定义和派生拥塞控制类。

➢ **传输可靠性机制**:与 TCP 类似,依靠包序号、ACK 序号、接收端 ACK 响应和丢包报告、丢包重传等机制实现了传输可靠性。

➢ **支持数据流和数据报两种传输方式**:UDT socket 支持 SOCK_Stream 和 SOCK_Dgram 两种类型,前者可靠,后者部分可靠。

➢ **支持点对点、防火墙穿透等穿透特性**:这一点与 UDP 是一样的。

➢ **面向连接的保活机制**:UDT 是基于握手机制(Shakehand)、心跳保活机制(Keepalive)和连接关闭机制的逻辑通道。

➢ **基于定时器进行数据包发送和确认**:与 TCP 需要等待 ACK 确认包后再行发送不同,UDT 基于定时器进行发送,也基于定时器实现 UDT 包确认。

➢ **不同的流量控制机制**:在拥塞窗口大小和当前可用的接收缓冲区大小之间选取最小值作为发送窗口大小,尽力确保发送和接收都不溢出。

➢ **支持带宽估计特性**:使用对包(Packet Pair)机制估计带宽。

- 对包机制需要发送端连续发送两个相同大小的包,接收端收到后会根据包的间隔来计算链路带宽,因为发送时每 16 个包为一组,所以只有最后两个是对包(即 16 号和 17 号包)。
- 发送端无需等到下一个发送周期再发送,接收端接收到对包后记录到达时间,并结合上次记录的值计算出链路的带宽,这个带宽会在下一个 ACK 确认包中反馈。

1. UDT 的软件架构

UDT 的软件架构如图 23-37 所示。可以看出,UDT 是基于 UDP 的,且 UDT 的支撑库是基于操作系统 socket 机制的,也就是说对 UDP socket 做了一层封装形成了 UDT socket 以供应用进程调用。拥塞控制(Congestion Control,CC)模块与 UDT socket 和 UDT 实现库进行交互。

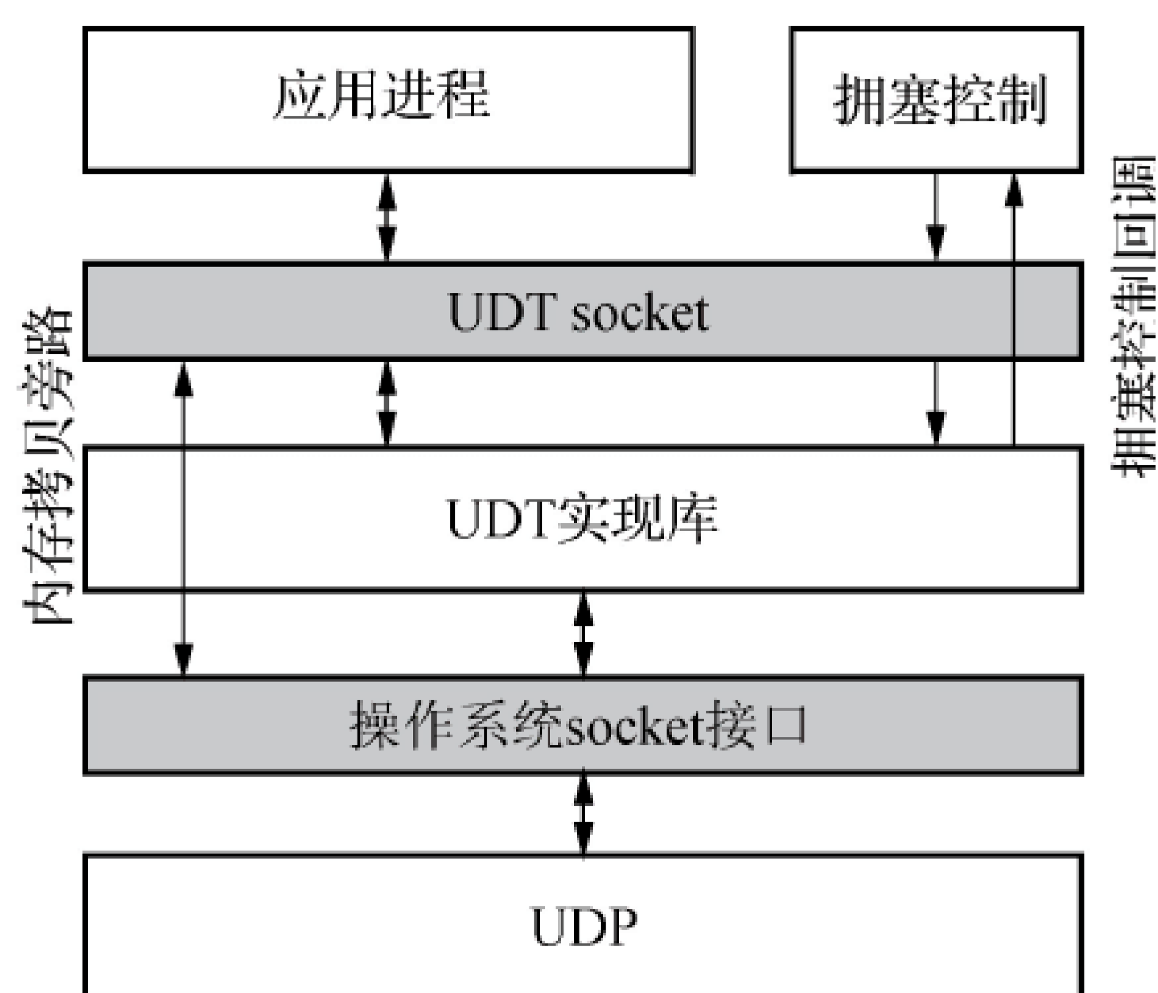


图 23-37 UDT 的软件架构

2. UDT 的接收与发送

➢ **数据发送**:当发送数据时,待发送的数据被拷贝到 Sender 缓冲区,Sender 再将其发送给 UDP 通道。

➢ **数据接收**:Receiver 从底层 UDP 通道获取数据包并拷贝到 Receiver 缓冲区。此时要对数据进行重新排序(Rerank)以查看是否有数据包丢失。Receiver 也会处理控制包,当接收到 NAK 控制报文后会更新 Receiver 和 Sender 的 LostList 并触发相应的事件(如拥塞控制等)。

从图 23－38 可以看出：拥塞控制模块与 Sender 和 Receiver 交互，Receiver 负责触发和处理所有的控制事件（如拥塞控制、可靠性控制等），并将这些事件回调给拥塞控制模块，以便于对 Sender 进行流量控制。拥塞控制模块包含了面向用户的回调函数集合，用于处理上述不同的控制事件。为了减少处理时间，内存拷贝绕过了 UDT 支撑层模块，而是直接发生于 UDP socket 和 UDT socket 之间。

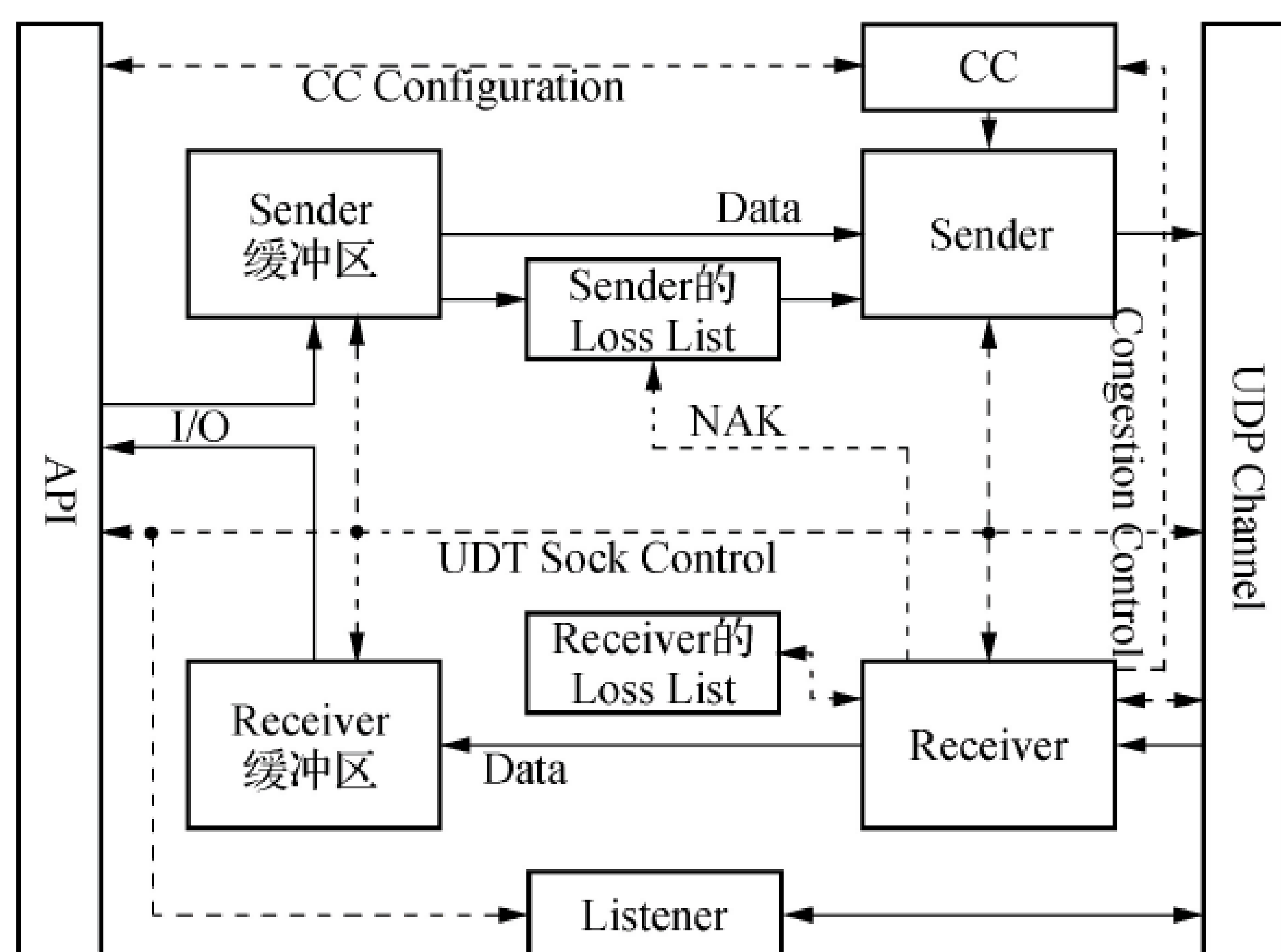


图 23－38　UDT 的收发流程

Lost List 用于存储丢失的包的序列号，这些序列号来源于接收端发送的 NAK 包的压缩信息或者超时事件里的超时包序号。因此，与 TCP 采用 ACK 包确认已收到包的机制不同，UDT 采用 NAK 包来确认未收到的包（并不代表 UDT 不使用 ACK 包）。

UDT 使用 ACK 和 NAK 机制来正向和反向激励发送行为。例如在收到 ACK 包时意味着正向激励，此时可以提高发送速度；反之，如果收到了 NAK 包，则意味着反向激励（有些包没收到，也许是因为拥塞导致的丢包），此时应该降低发送速度。当然 UDT 的调整不会在收到 ACK/NAK 包时立即执行，而是在每 10 ms 触发一次的定时器事件中执行的。

3．UDT 的连接与关闭

UDT 的连接分为两种模式，即 C/S 模式和会合连接（Rendezvous Connection）模式。

1）C/S 模式

这种方式类似于 TCP 方式，即启动监听、主动连接和被动接受的方式，其步骤如下：

（1）作为服务端的 UDT 实体首先启动监听。此时服务端可以接受和处理对端 UDT 实体的连接请求，并为新创连接生成一个新的 UDT socket。

（2）客户端连接服务端。首先发送握手请求控制报文，客户端以固定周期的方式发送握手请求控制报文，直到接收到服务端的握手请求回复或者超时结束。

（3）服务端接收到客户端的连接请求后，根据密钥和客户端的地址产生一个 Cookie，并将该 Cookie 返回给客户端，后续客户端的所有请求都要携带该 Cookie。

（4）服务端接收到步骤（3）返回的携带正确 Cookie 的握手请求控制报文后会分别比较

以下两组值：

- 握手报文数据包的大小与自身设置的数据包大小
- 握手报文滑动窗口大小与自身设置的滑动窗口大小

之后取它们之中的最小值保留为实际使用值，再连同服务端版本号、初始数据包序列号等一并返回给客户端。

（5）完成握手响应后，服务端就准备传输数据了。

- 在传输过程中，若收到从同一个客户端发过来的保持连接的握手请求，服务端就必须回复应答控制报文（客户端也一样）。
- 客户端接收到服务端的握手应答控制报文后也会准备传输数据。
- 另外客户端也会检查服务端是否和期望接受请求的服务端一致。

如图 23－39 所示是握手请求控制报文所携带的内容。

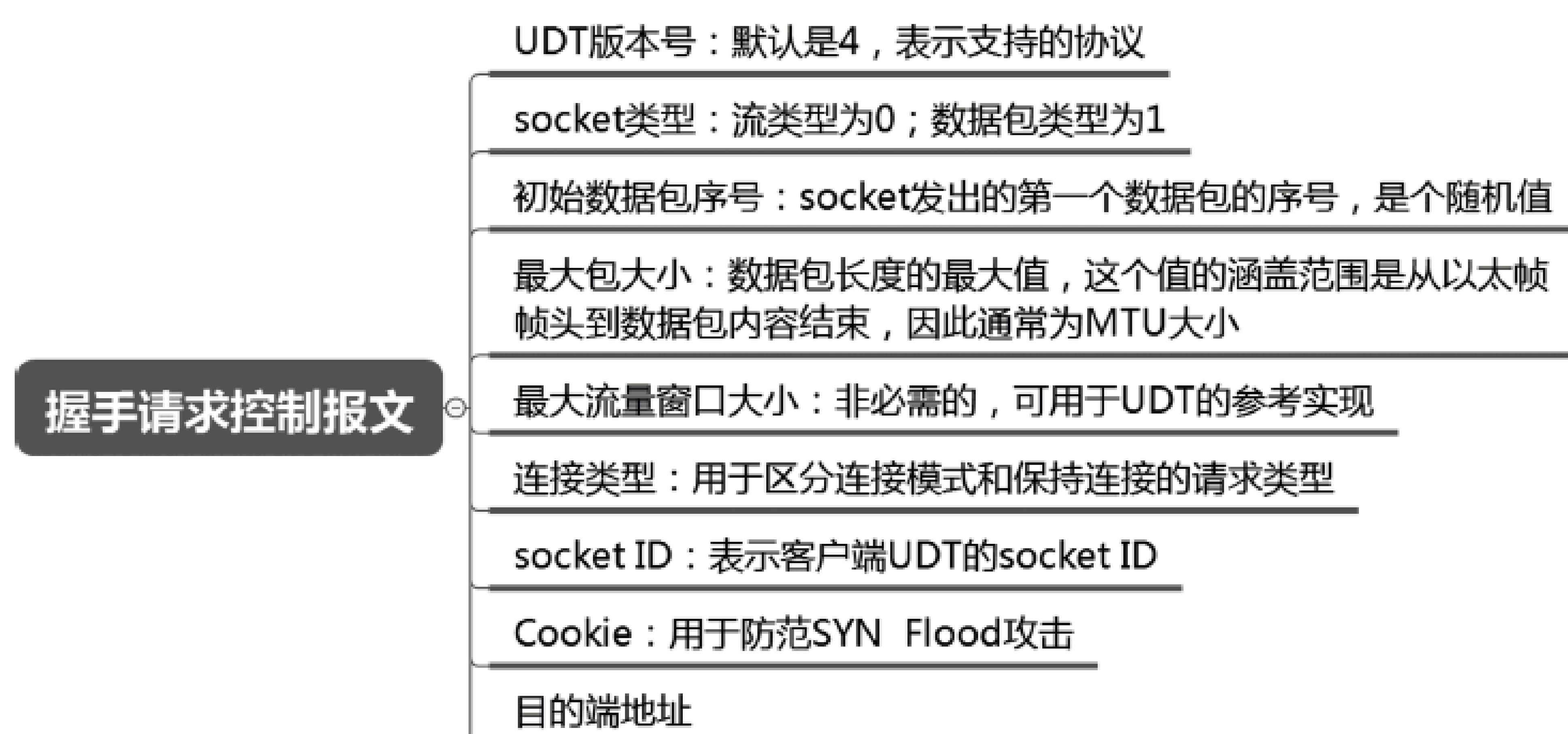


图 23－39　握手请求控制报文所携带的内容

2）会合连接模式

会合连接模式也被称为聚合模式或 P2P 模式，在这种模式下通信双方都是客户端，都会主动向对端发起连接请求，而每一端也都有能力接受连接请求。因此这种方式有利于私网穿透或防火墙穿透。其过程如下：

（1）通信双方同时向对端发起连接请求，发送握手请求控制报文。

（2）各端对收到的握手请求控制报文进行检查，检查的内容和过程如 C/S 模式中的步骤（4）和（5），各端只处理对端发过来的请求，不相关方发过来的连接请求会被忽略。

（3）验证检查通过后初始化连接。

会合连接模式下没有服务端的说法，各端都是客户端，都是对等的。

UDT 关闭的步骤就比较简单了。当已经建立 UDT 连接的一个实体关闭连接时，它将向对端实体发送一个关闭消息，目标 UDT 实体在接收到关闭消息后也关闭连接。另外，如果目标实体在连续 16 个 EXP 超时事件之后仍未收到关闭连接的消息，则会自己主动关闭连接。

4. UDT 定时器

UDT 定时器分为 4 种类型，用于触发不同的周期性事件，而且这些事件是相对独立的，也都以系统时间作为参考源。UDT 定时器是与 UDT socket 绑定对应的，这 4 种定时器分别为：

1）ACK 定时器

这种定时器用于触发接收端的 ACK 事件，其时钟周期是由拥塞控制模块设置的。但即使拥塞控制不需要基于定时器的 ACK，UDT 也会在不超过 10 ms 的间隔时间内发送一个 ACK 包。这个 10 ms 被定义为 SYN 时间（或叫同步时间），且这个时间影响着其他的 UDT 定时器。

2）NAK 定时器

这种定时器用于触发接收端的否定应答事件。NAK 定时器的时钟周期会动态地更新。

3）EXP（Expire）定时器

这种定时器用于触发接收端的数据包重传请求以及连接状态维护，其时钟周期也会动态更新。

4）SND（Send）定时器

这种定时器用于发送端处理基于速率机制的包发送。UDT 调整发送速率的固定周期一般为 10 ms，当收到 ACK 包时会认为是正向激励从而提升发送速度，当收到 NAK 包时会认为是反向激励进而降低发送速度。

5. UDT 多路复用

UDT 的多路复用通俗地说就是多个并发的 UDT 连接共享一个 UDP 端口。每个连接根据 UDT 包头的 socket ID 来区分，接收到的具有不同 socket ID 的 UDT 包会被分发到对应的 UDT socket 中。UDT 维护了发送和接收两个队列。

1）发送队列

发送队列包含了计划发送信息包的 socket 链表，这个链表上的每个 socket 必须包含至少一个计划发送的信息包（如果 socket 不包含计划发送的信息包就没有必要加入发送队列中了），且 socket 按照将要发送的信息包的发送时间进行排序。

发送队列也维护了一个高精度的定时器，当 socket 所包含的信息包的发送时间到达时，队列头部的 socket 包会被发送，之后便会从头部将该 socket 删除。当然如果 socket 包含了多个计划发送的信息包，则该 socket 又会被按发送时间的顺序重新插入发送队列里。

2）接收队列

接收队列读取收到的信息包并分发给对应的 UDT socket，如果目标 socket ID 是 0，则信息包将被分发给所有当前正在监听的 socket。

与发送队列类似，接收队列也包含了等待接收信息包的 socket 链表。如果队列中的每一个 socket 的 SYN 间隔时间到期，接收队列会检测该 socket 的每一个定时器是否超时。

6. UDT 包的结构

UDT 包承载于 UDP 包之上，是 UDP 包的负载。UDT 包分为两类：控制包和数据包，这两类包是通过包头的第一位来区分的，控制包包头的第一位为 1，数据包包头的第一位为 0。

1）UDT 控制包的结构

UDT 控制包的结构如图 23－40 所示。

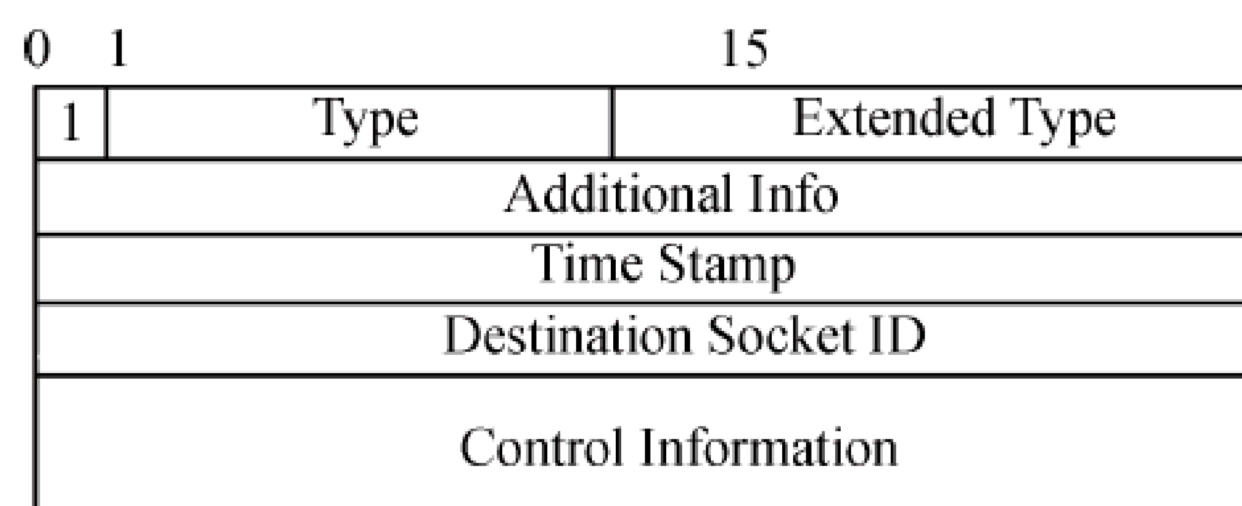


图 23－40　UDT 控制包结构

➢ **Type**：占 15 位，存在 8 种有效的控制包类型，包括：

- 0x0——UDT 握手协商报文，在这种报文中 Additional Info 未定义，但 Control Info 做了定义，即上文中所提到的握手请求控制报文所携带的内容。
- 0x1——UDT 心跳保活报文，在这种报文中 Additional Info 和 Control Info 都未定义。
- 0x2——UDT 的 ACK 应答报文，Additional Info 代表了 ACK 的 Sequence Number，Control Info 则包含了其他一些信息选项，例如可用字节数、收包速率等。
- 0x3——UDT 的 NAK 应答报文，Additional Info 未定义，Control Info 则包含了一些信息选项。
- 0x4——未被使用。
- 0x5——UDT 的关闭连接报文，Additional Info 和 Control Info 都未定义。
- 0x6——UDT 应答的应答（ACK2）报文，即对 ACK 的确认，ACK 发送时间与 ACK2 的接收时间可以用于估算 RTT。Additional Info 代表了 ACK 的 Sequence Number，Control Info 未定义。
- 0x7——UDT 的报文丢弃请求报文，Additional Info 代表了 Message ID（消息 ID），Control Info 则代表了报文的序列号。

➢ **Extended Type**：该字段的内容取决于 Type 字段的控制包的类型。

➢ **Time Stamp**：UDT 的相对时间戳，表示相对于 UDT 连接建立时所消耗的时间。该域是可选的。

➢ **Destination Socket ID**：UDT 的目标 socket ID，用于 UDT 的多路复用，多个 socket 被绑定到一个 UDP 端口上，该域用于区分不同的 UDT 连接。

2）UDT 数据包的结构

UDT 数据包的结构如图 23－41 所示。

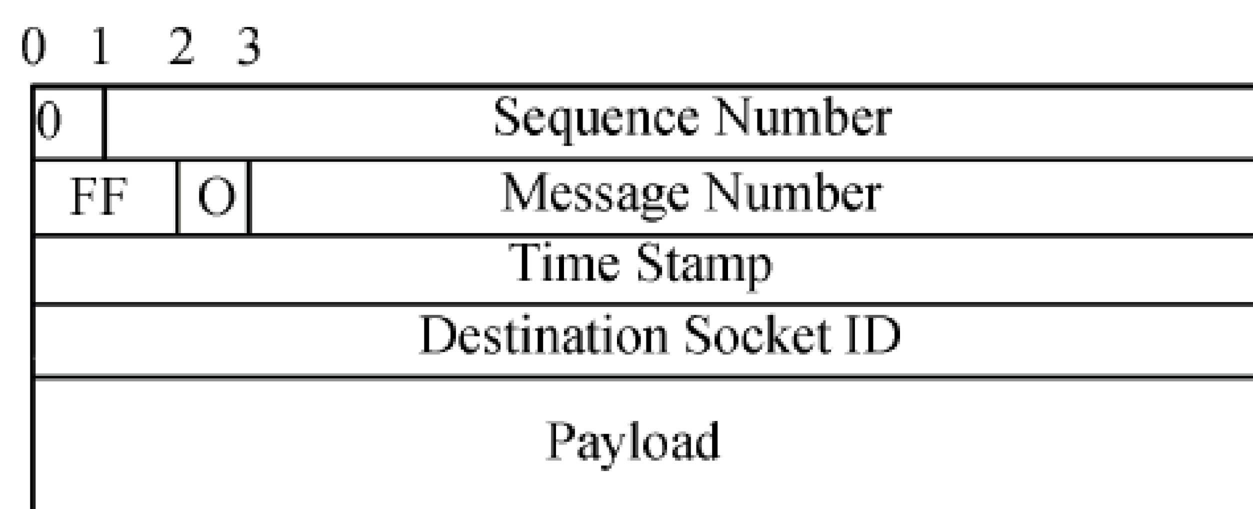


图 23－41 UDT 数据包的结构

- **Sequence Number**：UDT 数据包的序列号，每次发包时该序列号累加 1。
- **Message Number**：该域的前两位（FF）标记了当前包在整个消息包中的位置：
  - 10——表示当前包是消息的第一个包。
  - 01——表示当前包是消息的最后一个包。
  - 11——表示当前包是消息的唯一一个包。
  - 00——表示当前包是消息的中间包（不在头也不在尾）。

Message Number 的第三位表示数据包应该以顺序（1）还是以乱序（0）来投递，所谓顺序投递，即当前包投递时位于它之前的排队包都已被投递或丢弃。Message Number 的后 29 位表示消息的序列号。

- **Payload**：表示 UDT 包的负载。

23.2.4 SSL/TLS/DTLS

本小节我们着重讲述传输层的安全机制及其最常用的三种标准协议：SSL、TLS 和 DTLS。

SSL（Secure Sockets Layer，安全套接字层）是一种确保互联网上连接安全、保护敏感数据的标准技术。SSL 基于 TCP，是一种介于传输层（TCP）与应用层之间的安全协议。一般来说将 SSL（TLS 等）归为传输层协议。目前 SSL 的最高版本是 SSLv3.0。

SSL 协议的框架结构如图 23－42 所示。



图 23－42 SSL 协议的框架结构

随着 SSL 的演进，当更新到 SSLv3.0 版本时，IETF 对这个版本的 SSL 进行了标准化和有限度的增强，并将这个增强版本更名为 TLS 1.0。TLS 又被称为传输层安全（Transport Layer Security）协议，因此可以说 TLS 就是 SSL 的 3.0 增强版本（SSLv3.1），两者的框架结构也是一致的。我们研究 TLS 其实就是研究 SSL 的 3.0 增强版本，因此本节会着重介绍 TLS，也就

是 SSL 的增强版本，而 SSL 所用到的框架和流程的图示对于 TLS 也同样适用。

TLS 分为 1.0、1.1 和 1.2 三个版本，默认使用 TLSv1.0。SSL/TLS 最常见的应用是 HTTP 通信场景，HTTPS 实际上就是 HTTP over SSL/TLS。

无论是 SSL 还是 TLS 都是基于 TCP 的，因此业界迫切需要补齐针对 UDP 的安全套接字层的缺失，在这种情况下 DTLS（Datagram Transport Layer Security，数据报传输层安全）协议便被提出来了。其架构与 TLS 基本一致，只是 DTLS 是基于 UDP 的，除此之外包括握手交互、数据加密、身份认证等功能基本与 TLS 一致。

1. TLS

SSL/TLS 协议在传输层上封装了应用层的数据，这意味着应用进程模块（例如 HTTP 等）基本不需要做修改，在调用 socket API 的时候参数也基本不变。由于 SSL/TLS 是由自己完成身份认证与数据加密的，因此上层的应用进程也不用操心这些事。只是基于 SSL/TLS 进行传输的 HTTP 协议不再使用 80 端口而改用 443 端口了，这个端口也是 HTTPS 的"著名端口"。

SSL/TLS 协议从逻辑上可以分为两层：下层的 SSL/TLS 记录协议（SSL/TLS Record Protocol）层和上层的 SSL/TLS 握手协议（SSL/TLS Handshake Protocol）层，如图 23-43 所示。

- **SSL/TLS 记录协议层**：建立在 TCP 协议之上，为应用层协议提供数据封装、数据压缩、数据加密等安全类的功能支持。
- **SSL/TLS 握手协议层**：建立在记录协议层之上，用于在传输数据之前进行身份认证、加密算法协商、加密密钥交换等工作。

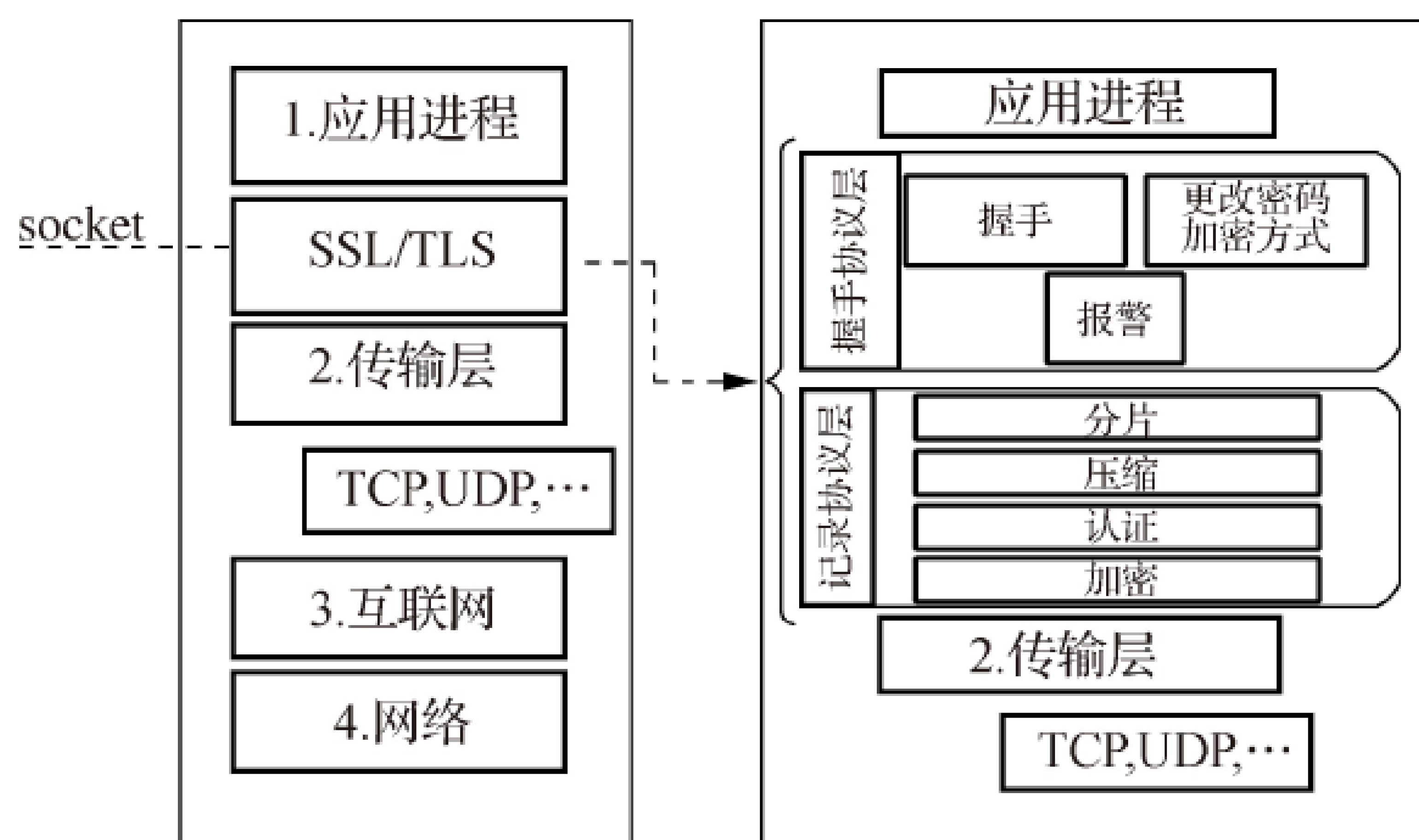


图 23-43 记录协议层和握手协议层在 SSL/TLS 协议中的位置

另外，SSL/TLS 协议还具有以下特性：

- 应用层协议可以透明地承载于 SSL/TLS 协议之上而无需做什么改动。
- SSL/TLS 记录协议层用于封装和加密高层协议数据的密钥是由通信双方协商出来的。
- SSL/TLS 协议在正式传输数据之前需要进行身份认证、密钥协商和密钥交换等工作，这些工作是通过握手协议完成的，并实现了以下目标：

- 保密性:保密性是通过加密技术实现的,第三方无法破解。
- 完整性:通过 MAC 校验机制可以立即发现报文篡改。MAC(Message Authentication Code,消息认证码)就是带密钥的散列函数。MAC 是基于密钥和消息摘要所获得的一个值,用于数据源认证和完整性校验。
- 防伪性:对通信双方都进行认证,双方都可以配备证书,以防身份被冒充。

SSL/TLS 协议一般采用对称加密算法(例如 DES、RC4 等),这主要是基于性能考虑的。密钥的协商则是在握手阶段完成的。当然,数据是否加密这是记录协议层要考虑的,可以加密也可以不加密,这完全视安全等级的需要而定。但无论是否加密,记录协议层对于数据都是要附加 MAC 的,一般将其附加在数据段的段尾,如图 23-44 所示。增加了 MAC 的数据可以保证不被篡改,同时也可以考虑是否需要整体加密。

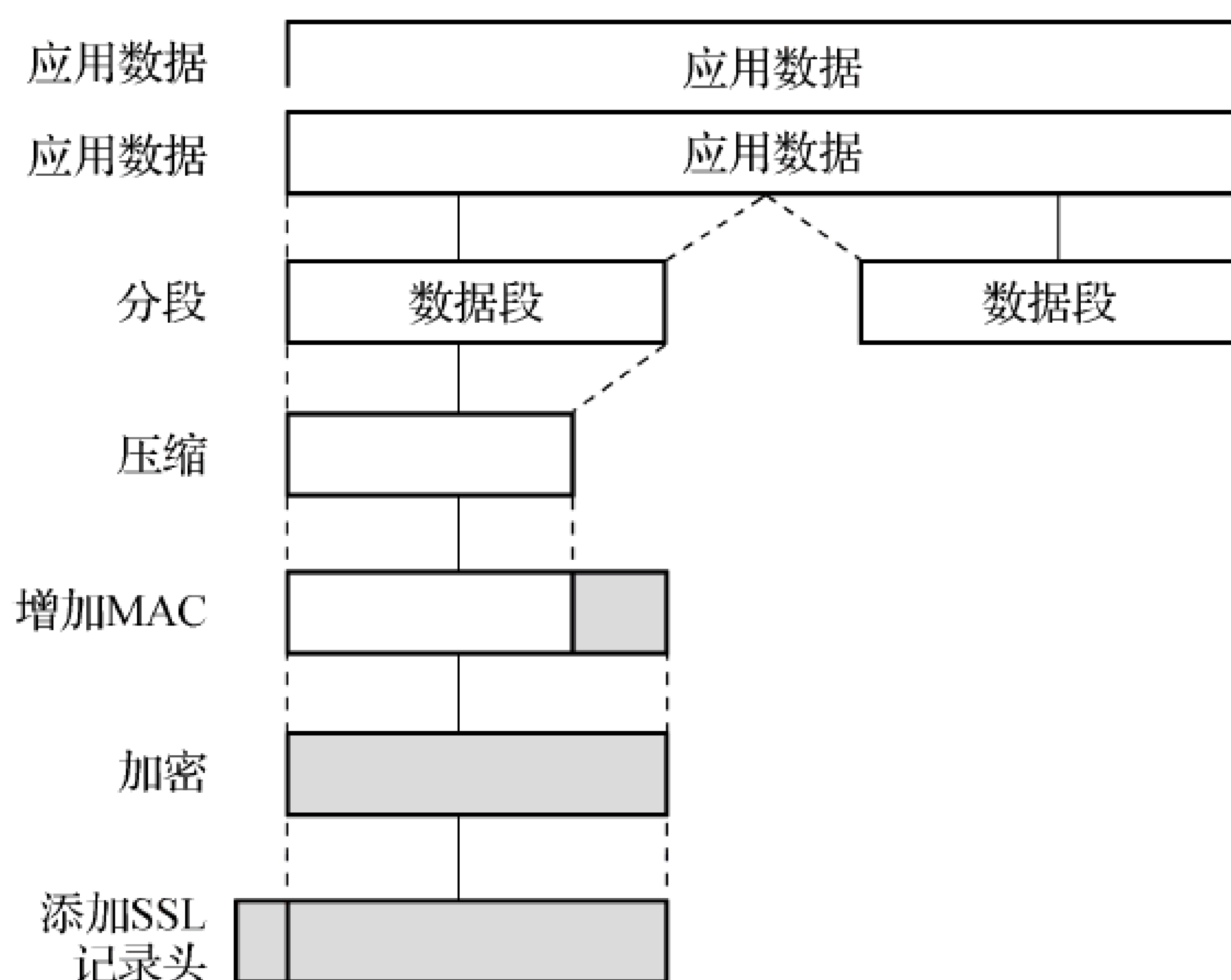


图 23-44　SSL/TLS 记录协议层对于应用层数据的处理

SSL/TLS 的握手协商是一件步骤繁复的事情。我们以 TLS 的协商为例(如图 23-45 所示)可以看到,从握手到传输数据再到关闭连接总共有十几个步骤,其中握手协商的步骤占了大多数。不过协商过程中有的步骤是可选的,可以根据通信的保密等级进行选择。

(1) 客户端发送 Client Hello 报文给服务端,该报文中包含了客户端支持的 SSL/TLS 的版本号、Session ID、支持的加密算法集合(Cipher Suite)、数据压缩方法(Compression Method)集合等信息。

(2) 服务端接收到 Client Hello 报文后回复 Server Hello 报文。收到 Client Hello 报文后服务端确定了能够支持的 SSL/TLS 协议版本、加密和压缩算法等,一并附着在 Server Hello 报文中返回给客户端。TLS 的 Hello 报文如图 23-46 所示。

(3) 这一步可选,服务端发送本端的数字证书给客户端(Send Certificate 报文)。

(4) 这一步也可选,服务端可能也想看看客户端的数字证书,这便是双向认证,因此向客户端发送证书请求报文(Request Certificate 报文)。

(5) 至此,服务端在握手协商过程中的使命也就基本完成了(虽然步骤(4)的客户端证

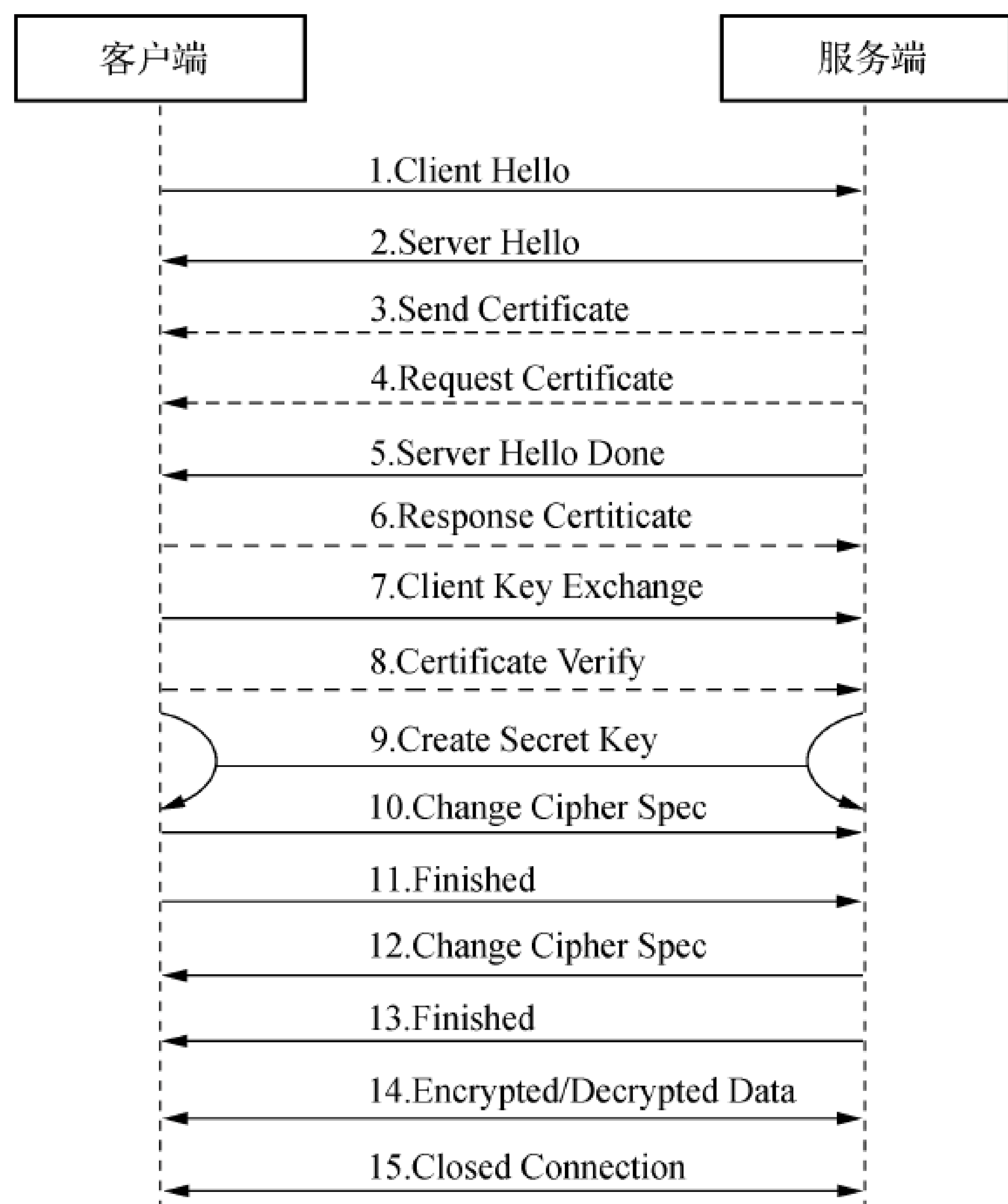


图 23－45　SSL/TLS 协议的握手协商流程(虚线为可选步骤)

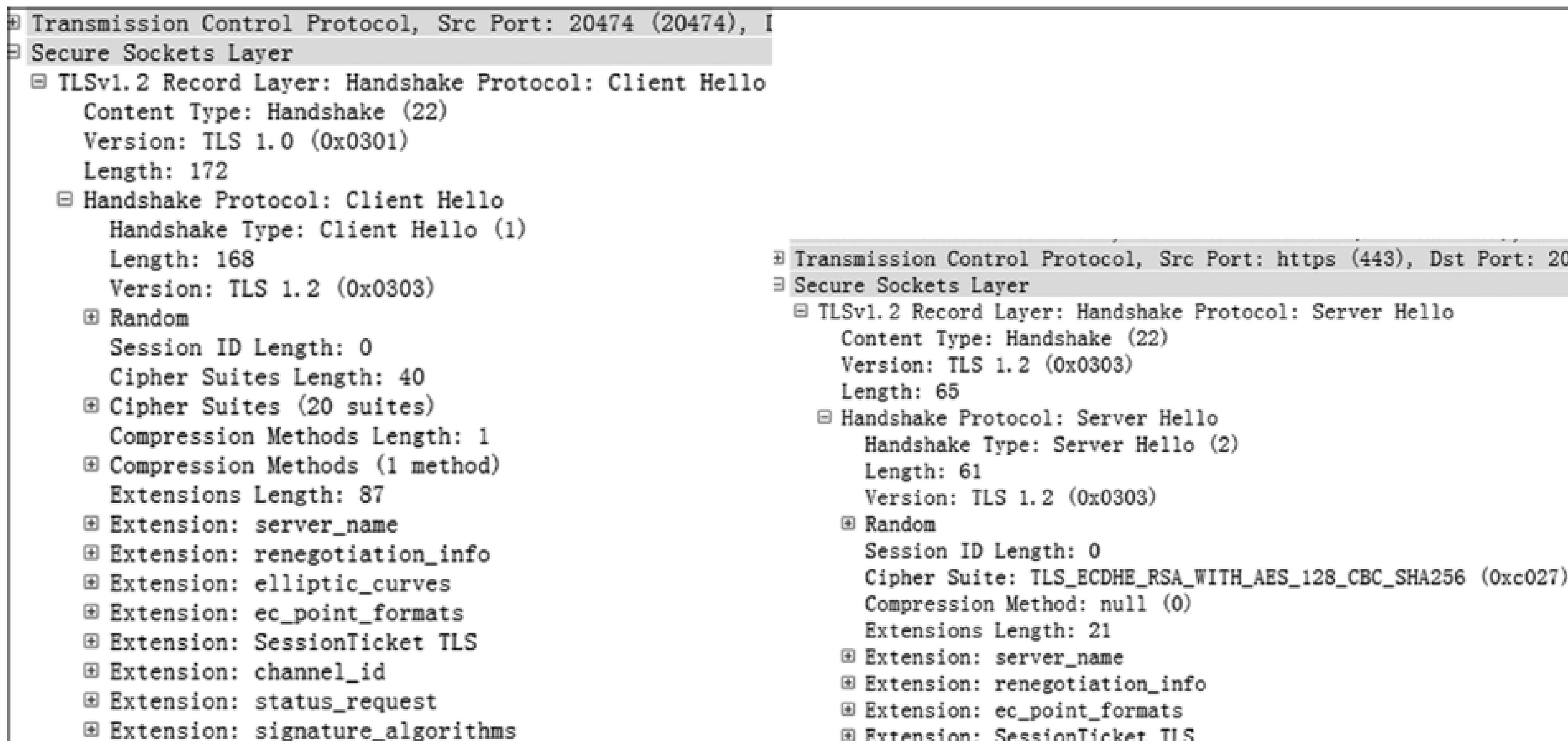


图 23－46　TLS 通信双方的 Hello 报文

书可能还没有返回),因此向客户端发送服务端初始协商结束通知(Server Hello Done 报文)。注意,这里只是初始协商结束,老鼠拉木锨,大头在后面呢。

(6) 这一步是客户端对步骤(4)的回应,因此也是可选的。在双向认证的情况下,客户端通过 Response Certificate 报文将本端的证书发送给服务端。

(7) 客户端发送 Client Key Exchange 报文,使用服务端的公钥对客户端公钥和密钥种子

进行加密并发送给服务端。

（8）本步骤可选，即当双方选择了双向认证的情况下，客户端用本地私钥生成本端证书的数字签名并发送给服务端（Certificate Verify 报文），以便服务端通过公钥解密并进行身份认证。

（9）通信双方通过密钥种子等信息生成通信会话密钥（Create Secret Key），这个密钥是对称式密钥，必须足够长，具有很高的破译门槛。

（10）客户端通过 Change Cipher Spec 报文通知服务端通信方式切换为加密模式。

（11）客户端向服务端发送 Finished 报文以通知自己做好了加密通信的准备。

（12）服务端通过 Change Cipher Spec 报文通知客户端通信方式切换为加密模式。

（13）服务端向客户端发送 Finished 报文以通知自己也做好了加密通信的准备。

（14）通信双方使用通信会话密钥对传输的数据进行加密和解密。

（15）通信结束后，任意一方都可以主动断开 SSL/TLS 连接（发送 Closed Connection 报文）。

从以上流程上可以看出，密钥协商的 4 个阶段分别是服务端的 Certificate（证书认证）和 Server Key Exchange（交换公钥和密钥种子）以及客户端的 Certificate 和 Client Key Exchange。这 4 个阶段的协议可以定制，包括消息的使用方法也可以更改，将来若要扩展密钥协商的方法就可以采用定制的方式实现。还要注意的是，协商阶段的多个步骤可能会被合并成一条报文传输以提高通信效率。

在 SSL/TLS 的握手协商流程中，非对称加密实现了身份认证和密钥协商，对称加密算法则主要用于对数据报文进行加密，而散列函数则用于验证信息的完整性，如图 23－47 所示。

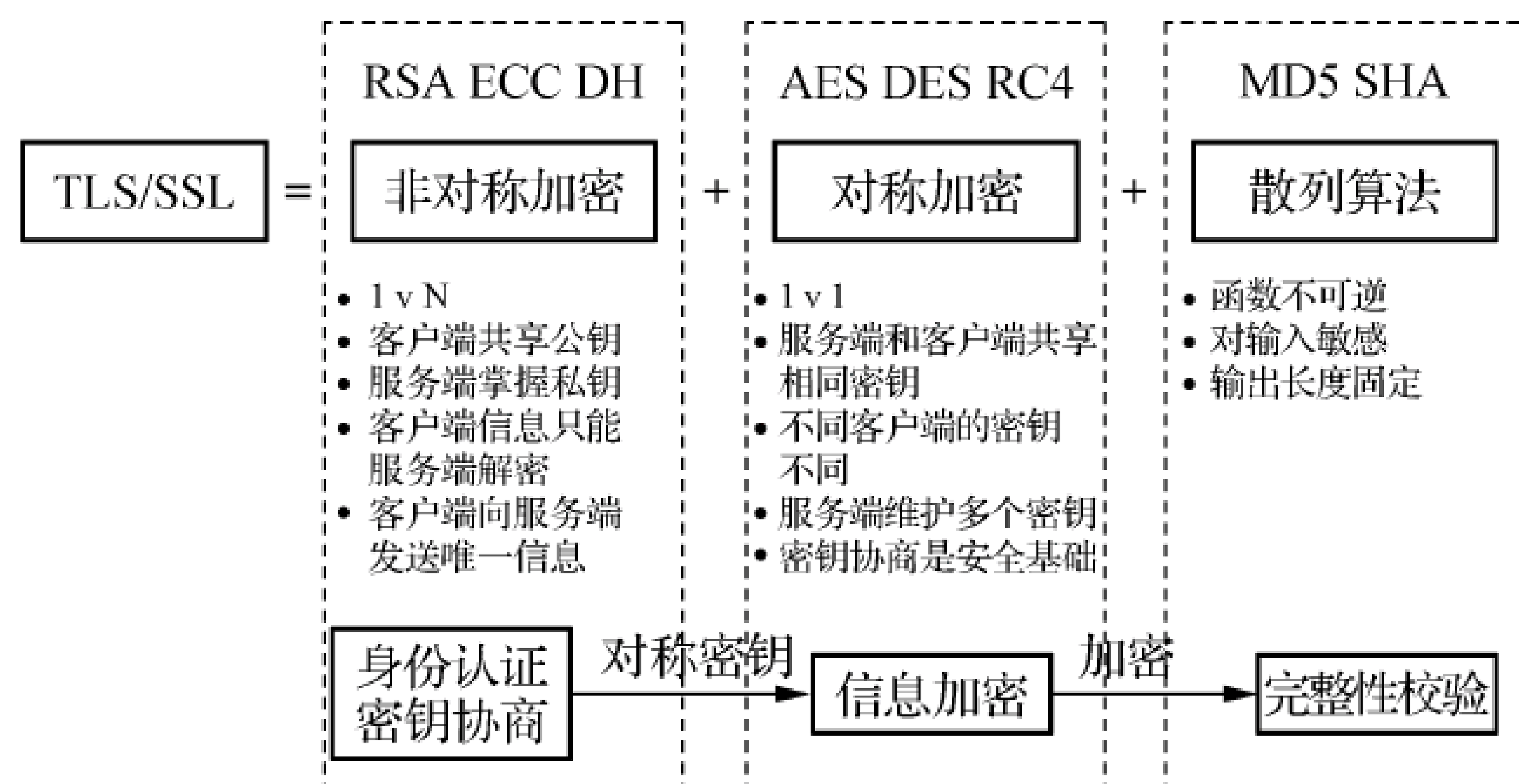


图 23－47　SSL/TLS 协议所采用的加密算法与散列算法

在 SSL/TLS 协议中还必须对连接的重连做出规定。例如当客户端访问了在服务端受到保护的数据时，或在双方更新通信密钥时，都会发生连接重连。前一种情况一般造成服务端主动重连，后一种情况一般造成客户端主动重连。

在客户端与服务端之间已经建立起有效的 SSL/TLS 连接并正常通信的情况下，若客户端访问了服务端受保护的数据，此时为了保护这些数据，服务端会要求重新认证，因此服务端会主动向客户端发送 Hello 报文以图重新建立连接，客户端收到服务端的 Hello 报文后也会回复 Hello 报文以同意重新建立连接，如图 23－48 所示。





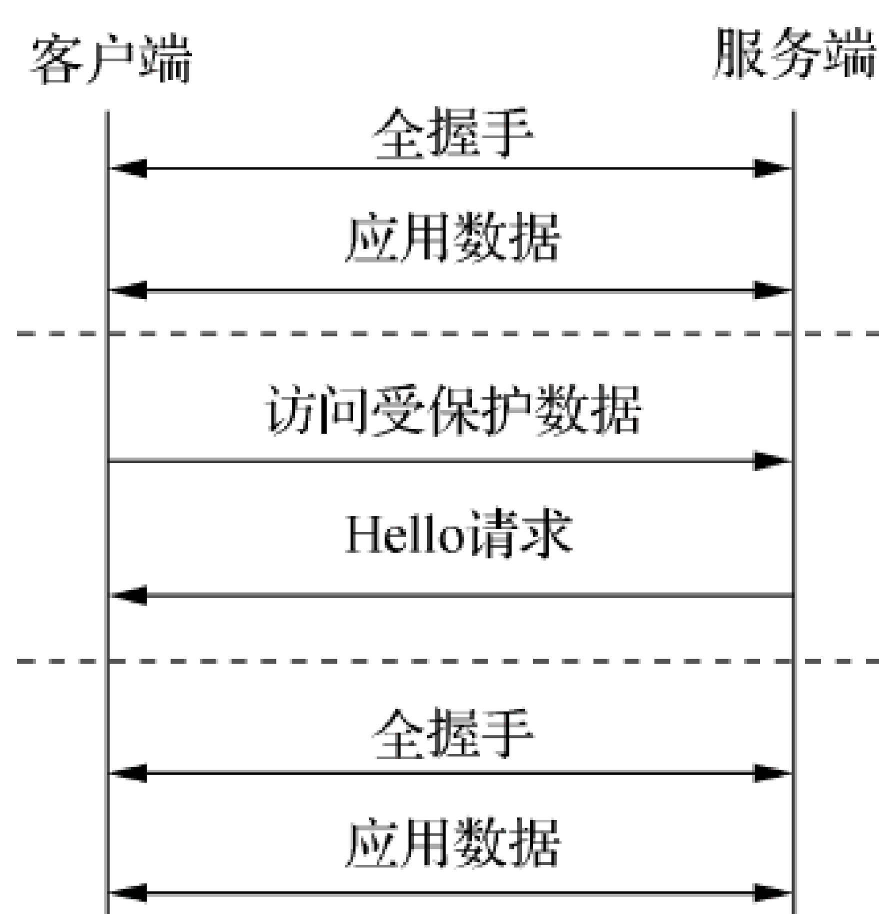


图 23－48　服务端主动重连

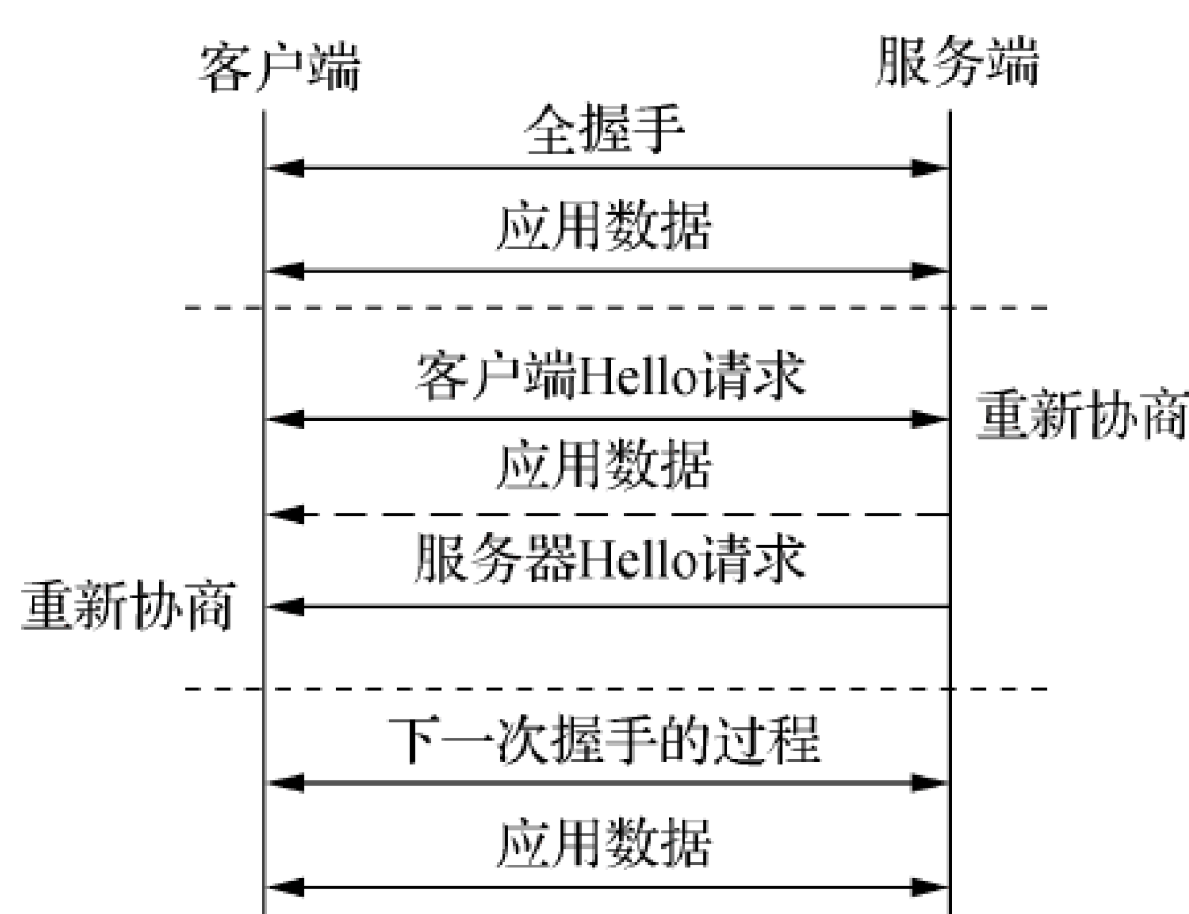


图 23－49　客户端主动重连

客户端主动重连的过程与上述类似，如图 23－49 所示。在通信过程中如果客户端感觉需要更新密钥，则会主动发出 Hello 报文。但由于 Hello 报文的内容无法被服务端立即识别出，因此服务端会将这个错误提交给上层继续处理，一般上层会要求重新建立连接以更新密钥。此时客户端已经停止向服务端发送数据了。服务端一旦得到上层的重新建立连接的指令就会返回 Hello 报文给客户端。当然客户端也无法立即识别，也会像服务端一样回调到上层并被要求重新建立连接，并由此开启连接重建的过程。

2. DTLS

作为基于 UDP 的传输层安全标准，DTLS 主要是为了弥补 TLS 的不足。但是 DTLS 在安全机制上对 TLS 的改进很少，并且沿用了 TLS 的大部分标准和实现代码。例如 DTLSv1.0 是基于 TLSv1.1 的，DTLSv1.2 则是基于 TLSv1.2 的。

DTLS 协议也分为记录协议层和握手协议层两个层次，其框架结构如图 23－50 所示。

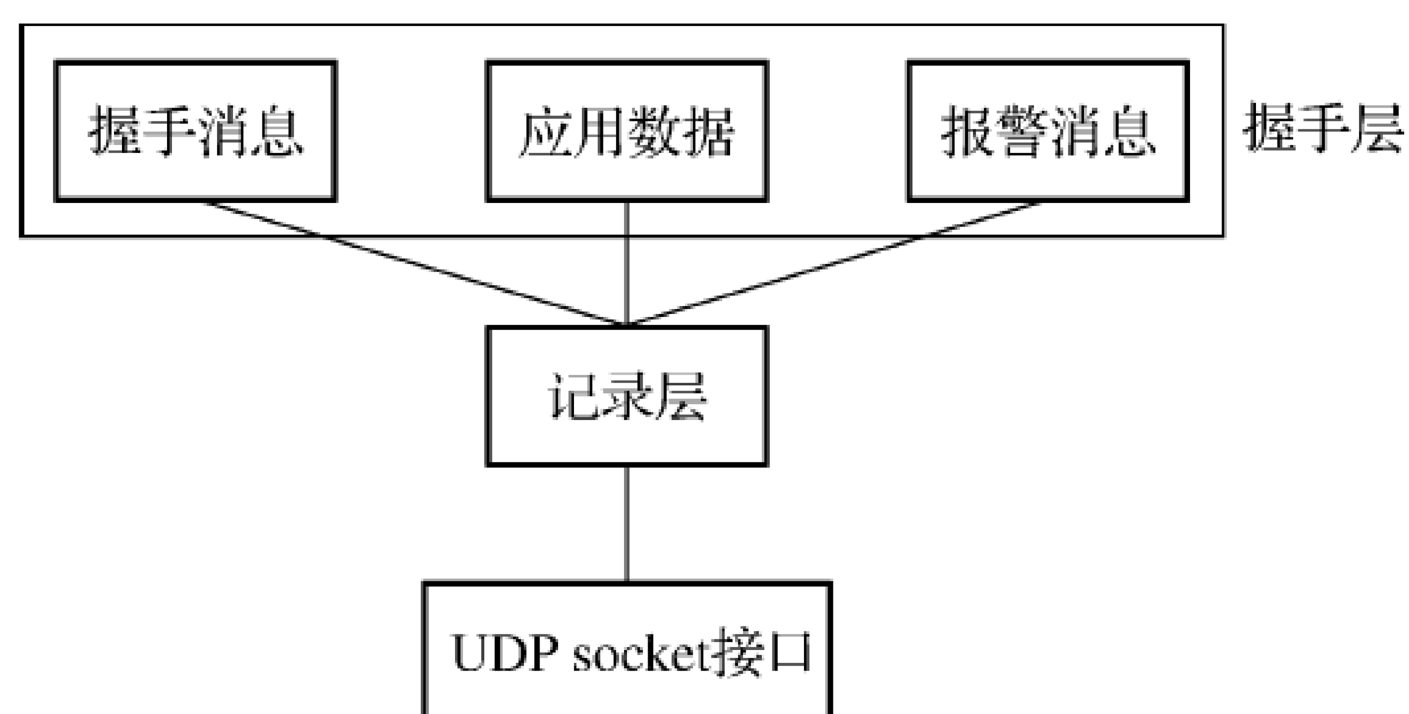


图 23－50　DTLS 协议的框架结构

从上图可以看出，DTLS 协议框架中的握手协议层主要包括握手消息、应用数据和报警消息。构造 DTLS 报文段时，首先产生上层的负载消息（握手消息、报警消息等），然后添加头部以构成上层消息全文。再以此作为记录协议层的负载，添加记录协议层的头部以构成完整的 DTLS 报文段，最后调用 UDP 的 socket 接口发送。在这个过程中，DTLS 的加密是针对记录协议层负载的（消息全文），在 DTLS 协议中就是 Finished 消息报文和应用数据报文两种类型。

接下来我们分析一下 DTLS 协议的握手协商流程，如图 23－51 所示。

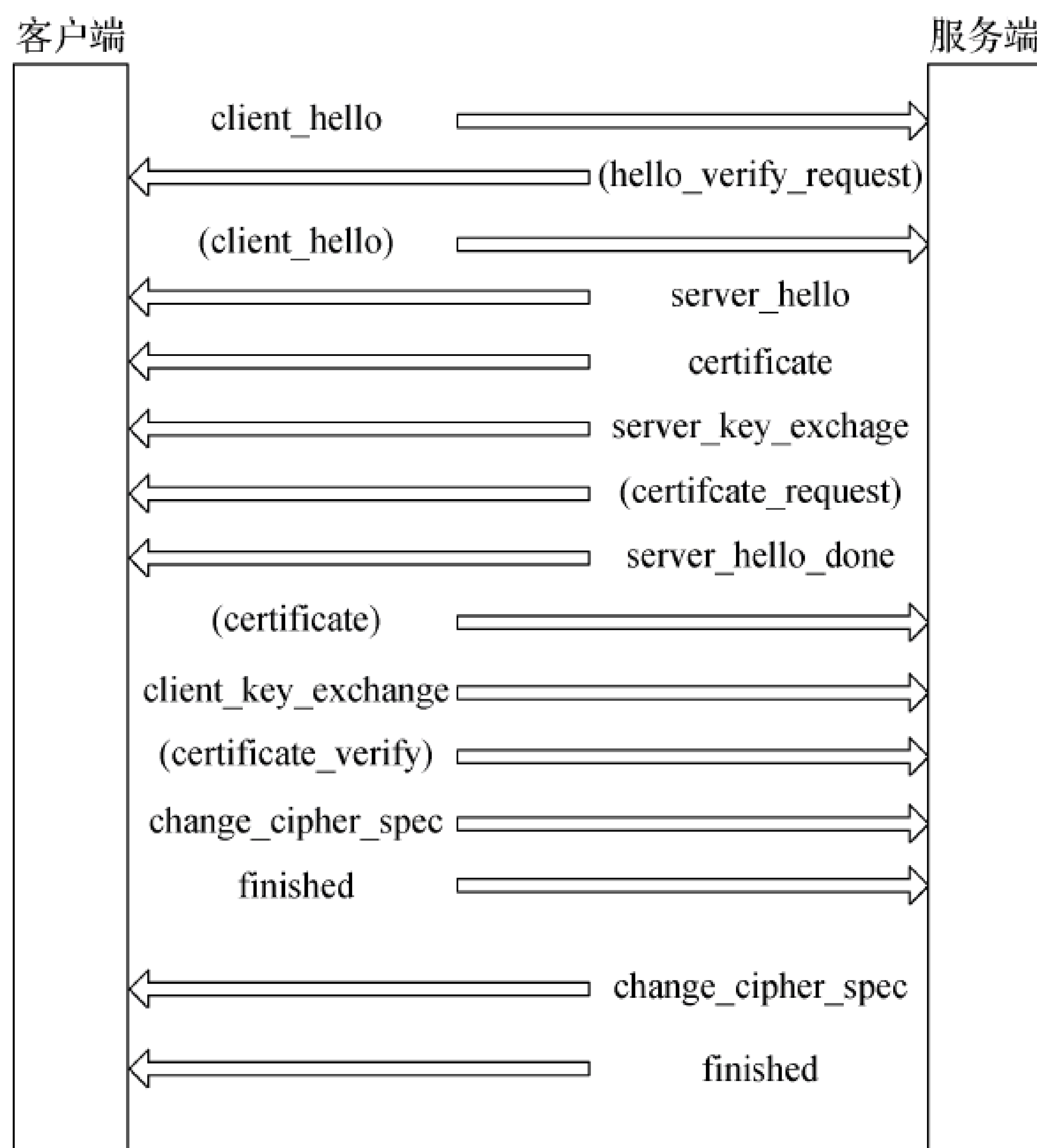


图 23－51　DTLS 协议的握手协商流程

DTLS 协议与 TLS 协议的握手过程基本一致，图 23－51 中括号内的为可选步骤。在 DTLS 协议的 Client_Hello 和 Server_Hello 报文中都包含了 32 字节的随机数、客户端支持的加密方式集合等，这些也与 TLS 如出一辙。在这些步骤中要注意以下几点：

➢ 客户端向服务端发起连接请求时，服务端可以根据事先的配置选择是否验证客户端的 Cookie 和证书，这分别对应了服务端发送的 Hello_Verify_Request 和 Certificate 请求报文。

➢ Certificate 就是证书报文，包括了服务端的证书。

➢ Server_Key_Exchange 报文包括了用于进行 ECDH 密钥协商交换的参数，以便协商出相同的密钥来加密会话的应用数据内容。为了防范中间人攻击，报文末也对整个报文进行了 ECDSA 签名。

➢ Server_Hello_Done 报文的负载为空，这条报文只是用作标志，表示服务端当前阶段的报文发送已经完毕。

➢ Change_Cipher_Spec 报文的负载也是空的，用于向服务端通知本端（客户端）已经计算出主密钥，之后发送的报文都是以主密钥加密的。

➢ Finished 报文是握手阶段的最后一个报文，包括了消息验证码等信息，这些信息也是用主密钥加密的。

DTLS 可能遭遇两种类型的 DoS 攻击，一种是资源消耗型，另一种是放大攻击型。所谓放大攻击就是仿冒被攻击者的 IP 地址向服务端发送初始化报文，以求服务端返回一个体积

很大的证书给被攻击者。这种请求多了就会造成被攻击者的瘫痪。可以通过 Cookie 验证机制减少遭遇放大攻击的可能性。

Cookie 验证机制要求客户端发送 Cookie,服务端可以验证其 IP 地址是否可信以减少 DoS 攻击的可能性。其具体做法是:

(1) 第一次的 Client_Hello 报文中含有 Cookie 一项,如果为空,则说明这条请求是要新建连接,服务端会根据客户端的源 IP 地址通过散列算法生成一个 Cookie 并填入 Client_Hello_Verify 报文中。

(2) 第二次的 Client_Hello 报文中带入了上述 Cookie 值,服务端会校验这个值的一致性以决定是否继续握手。

除此之外,DTLS 也支持重传机制和会话恢复机制。由于握手成功后服务端生成的 Session ID 会返回给客户端,因此客户端在下次连接时可以附带这个 Session ID。如果 Session ID 正确,则服务端允许客户端沿用原有的会话数据(例如协商算法和密钥等)。

23.2.5 TCP 的拥塞控制机制

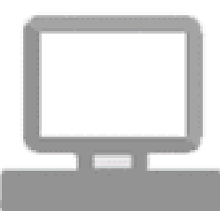
从前面的章节中可以看出,以 UDP 方式发送应用数据以求效率,以 TCP 方式对应用数据收发进行拥塞控制以求可靠,这是近年来新的传输层协议的特点,这些传输层协议统称为 RUDP。前面几个章节已经介绍了这些协议,在此我们介绍一下 TCP 的拥塞控制机制。

TCP 有两种比较核心的控制机制:拥塞控制和流量控制。TCP 的流量控制一般采用我们比较熟悉的滑动窗口机制实现,通过控制发送端的发送速率来匹配接收端的接收速率。TCP 中的拥塞一般是由链路中交换机或路由器等中间节点的缓冲资源不足造成的,也就是常说的"拥塞丢包"。我们说 TCP 是可靠协议,但并不意味着不丢包,IP 包在路由器上被挤丢了岂是发送端和接收端能控制的事?但是丢了可以补发,TCP 可靠的含义就在于此。并且这个补发也是上层应用软件无感的,是由 TCP 本身搞定的。特别是在长链路环境下,中间节点的带宽和缓冲资源更为匮乏,因此拥塞产生的几率更高。从这个角度讲,拥塞控制就是要防止过多的数据被注入到网络中,以减轻网络节点的负载,因此拥塞控制是个整体性和全局性的过程。

在介绍拥塞控制之前先来解释几个名词。

- **拥塞窗口(congestion window,cwnd)**:这是 TCP 发送端维护的一个窗口变量,描述了发送端在拥塞控制情况下一次能发送的最大数据量。在传输过程中这个值可以动态调整,代表了链路中设备的传输能力,也代表了网络的拥塞程度。
- **接收窗口(receive window,rwnd)**:用于对端通告,接收窗口值的大小代表了已发送出去但尚未收到 ACK 确认包的最大报文长度,接收窗口值越大数据传输速率越快。
- **滑动窗口(sliding window)**:这是由 TCP 接收端维护的用来进行流量控制的窗口变量,代表了 TCP 接收端的接收能力。
- **慢启动阈(ssthresh)**:这是一个为了限制 cwnd 无限增长而设置的阈值,是慢启动与拥





塞避免两个阶段的分界点。

- 当 cwnd < ssthresh 时,TCP 使用慢启动算法;
- 当 cwnd = ssthresh 时,TCP 既可以选择慢启动算法也可以选择拥塞避免算法;
- 当 cwnd > ssthresh 时,TCP 使用拥塞避免算法。

在这两个阶段,cwnd 是动态增长的,ssthresh 则是不变的,且初始值通常设置为 65 535 字节。

➢ **最大段长度(Max Segment Size,MSS)**:以太网 MTU 减去 IP 头与 TCP 头后剩下的字节就是 TCP 在一个 IP 包中的最大段长度。
- 如果发送窗口能够最大限度地满足 MSS 则传输利用率高。
- 以太网 MTU 为 1 500 字节,因此 MSS 最大可以为 1 460 字节。

➢ **回路响应时间(RTT)**:一个 TCP 数据包从发送端发送开始,到发送端接收到接收端返回的 ACK 确认报文的时间间隔。

➢ **重传超时时间(RTO)**:数据包从发送到失效的时间间隔,是判断数据包丢失与否及网络是否拥塞的重要参数,RTO 通常设置为 2RTT 或 5RTT。

➢ **全局同步**:传输链路中间节点(如路由器等)在缓存紧张时可能会丢弃报文分组,这种情况发生时的影响往往是全局性的。例如针对 TCP 连接分组的大面积丢弃会导致许多条连接产生拥塞。因此这么多条连接会同时进入慢启动阶段和快速恢复阶段,导致网络流量的大起大落、剧烈抖动。

TCP 的拥塞控制比较复杂,包括了慢启动(Slow Start)、拥塞避免(Congestion Avoidance)、快速重传(Fast Retransmit)和快速恢复(Fast Recovery)这几个阶段。

1)慢启动与拥塞避免

慢启动与拥塞避免的核心思想是:发送的初期阶段可以发送得稍微快一些,这个快是指发送数据量的增长速度,这种增长是乘法级的,这个过程叫“慢启动”或叫“慢开始”;发送的后期阶段要发送得稍微慢一些,发送数据量的增长速度要平缓,这种增长是加法级的,这个过程叫“拥塞避免”。当执行完这两个阶段后网络负载会达到峰值,再往后的话,如果控制得不好就会发生拥塞;控制得好,链路的利用率可以达到顶峰。如果发生拥塞,则再从慢启动开始执行(较新的 TCP 版本已经废除从慢启动阶段重新开始的策略),这是基于一种“大公无私”的信条:TCP 不是一个自私的协议,当拥塞发生的时候,要做自我牺牲。

在图 23-52 中,cwnd 代表了发送的数据量(这里是以 MSS 为单位的,实际在网络中以字节数为单位,确认的是数据段的偏移值),接收端每发送一个 ACK 确认包后 cwnd 都会增加一倍(窗口乘法级增长),这是理想状态下的慢启动过程,这个过程直到 cwnd 等于 ssthresh 时结束(见图 23-53 中慢启动与拥塞避免的转折点,即 ssthresh = 16 处)。



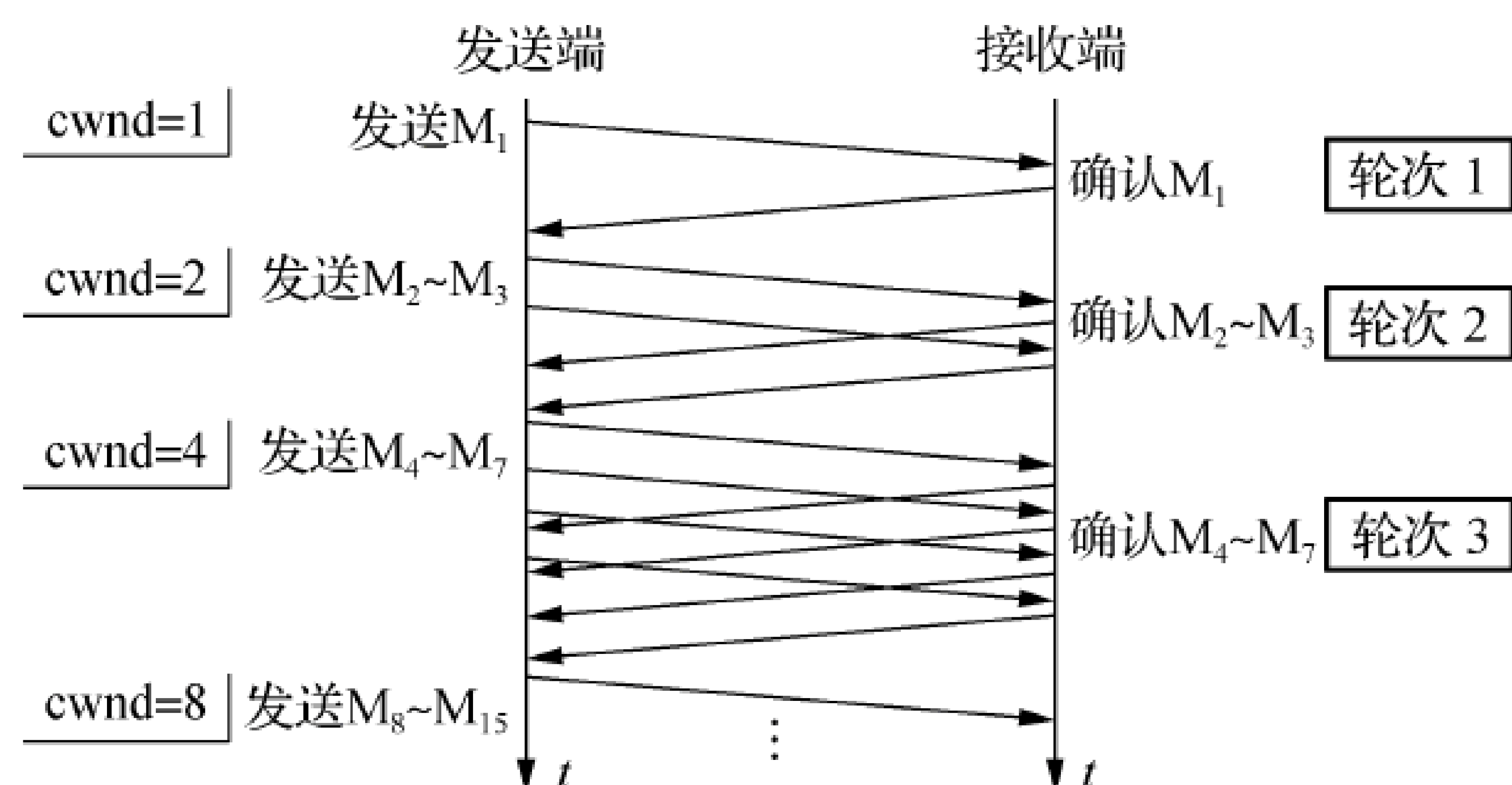


图 23－52　TCP 慢启动阶段中 cwnd 的增长过程

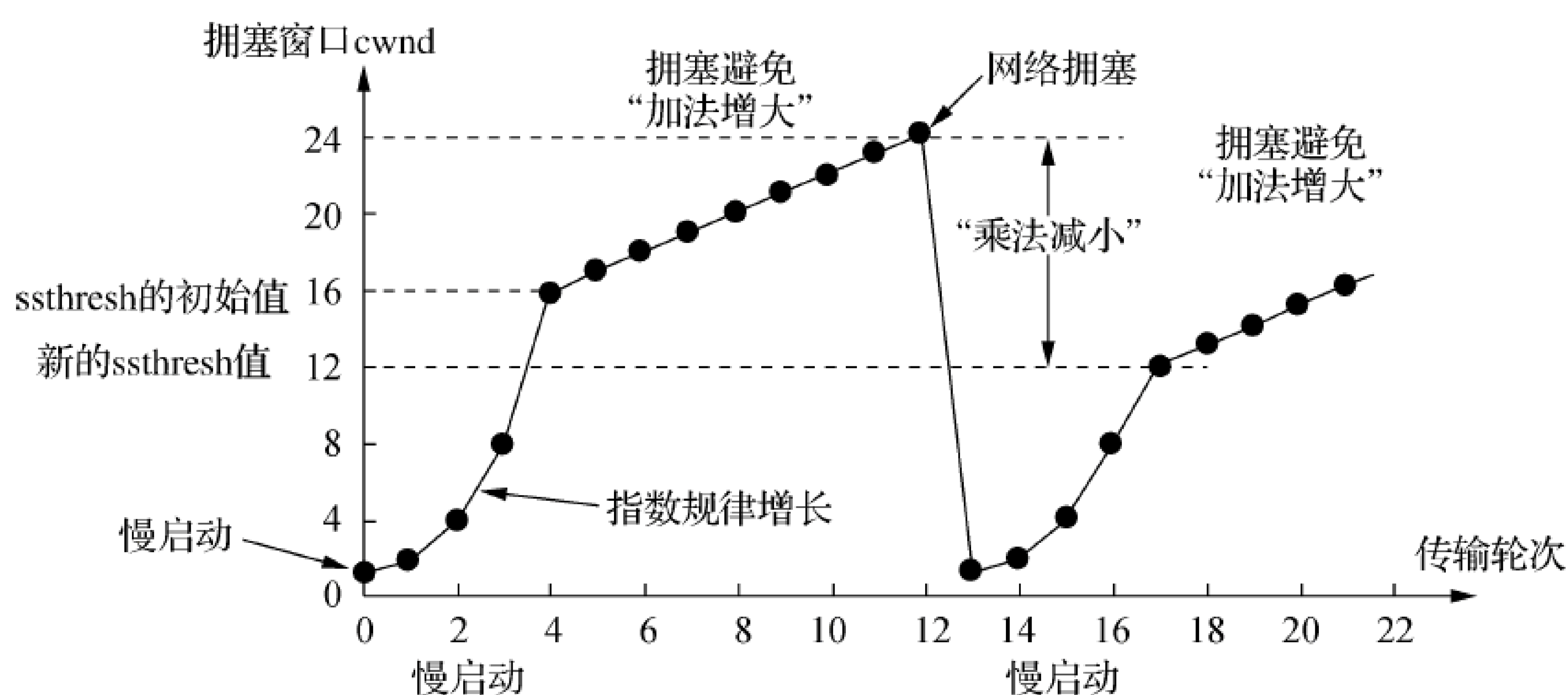


图 23－53　慢启动与拥塞避免的过程

慢启动过程后紧接着就是拥塞避免过程，临界点就是 cwnd = ssthresh。从这时起，cwnd 是加法级增长的，每过一个 RTT 后 cwnd 只增加 1。这是因为 cwnd 达到一定的阈值后如果继续激进地增大发送量则会大概率地引起拥塞。

但在拥塞避免阶段 cwnd 也是增长的，这个增长总是有个度的，这个度就是拥塞发生。那么判断拥塞发生的标准是什么？就是在一个 RTO（重传超时时间）内没有收到 ACK 确认包。这个判断标准非常简单也非常武断，不考虑因长肥链路距离过长 ACK 包还在途的情况，统统“错杀”了。但这也是无奈之举，因为接收端是没有能力分辨 ACK 包是丢失还是在途的。

当拥塞发生时，TCP 将 ssthresh 设置为当前 cwnd 大小的一半，而 cwnd 则设置为 1，再重新开始慢启动和拥塞避免的过程。

2）快速重传与快速恢复

我们先来介绍一下接收端收到失序报文时的做法。

所谓失序（Disorder）就是报文排序不规则，中间有缺失。TCP 是流式传输，以字节为单位，不存在报文序号的问题，这里说的是报文段的偏移值不连续，中间有缺失。如果接收端收到失序报文，它会向发送端连续发送三次重复的 ACK 确认报文（DUPACK），该报文中的确认号（ACK）为丢失报文段的序号。发送端收到连续三次这样的重复确认包就会知道这个

报文段丢失了，应该重传。接收端对失序报文的这种立即确认被称为“快速重传”，如图 23－54 所示。快速是相对重传计数器方式而言的。在采用重传计数器方式时，TCP 在等不到 ACK 确认报文的时候会重新发送上次 ACK 确认后发送的数据报文，但这种方式要等到 RTO 到期，与快速重传一比就太慢了。

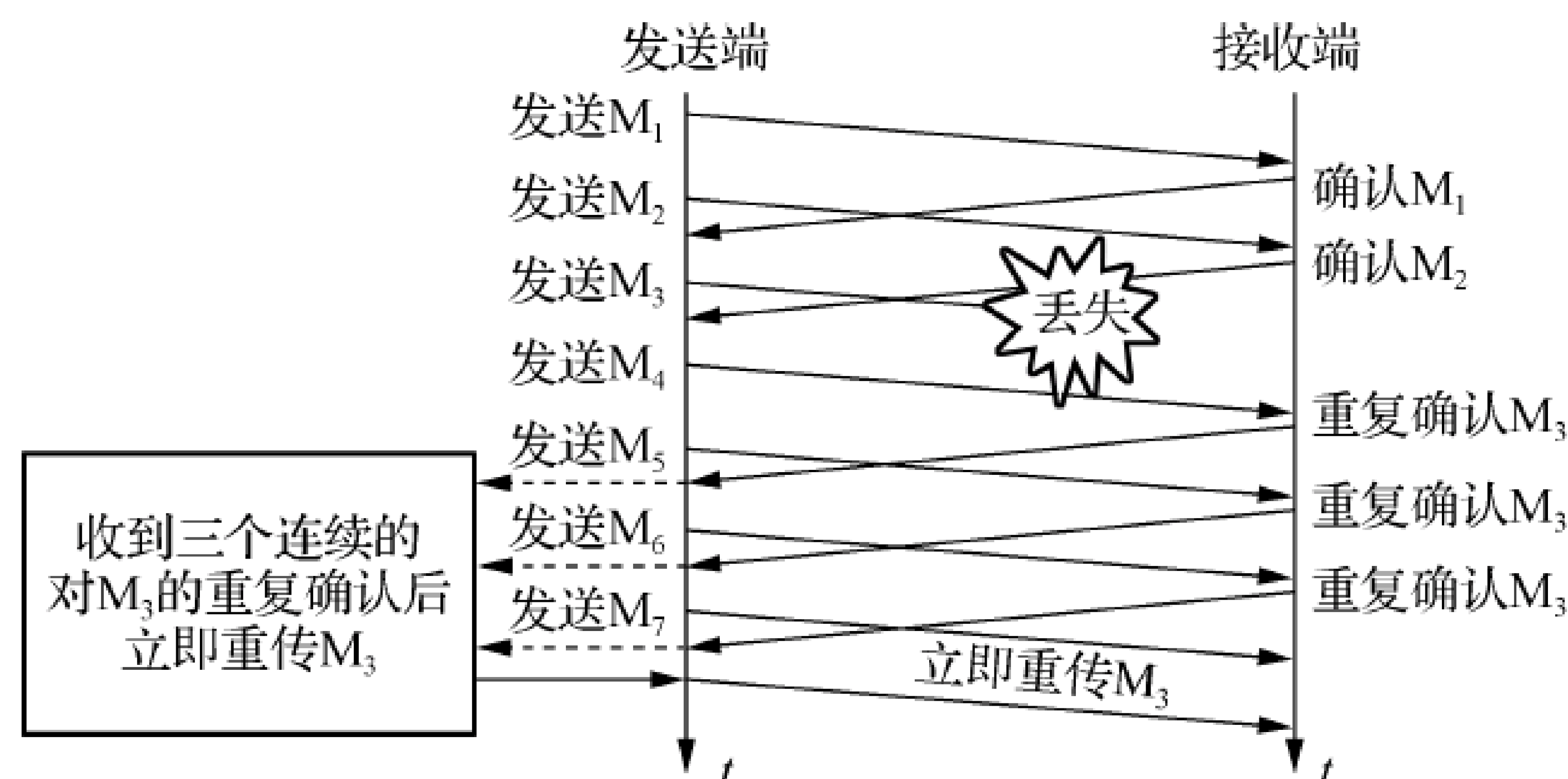


图 23－54　快速重传的过程

TCP 中快速重传算法和快速恢复算法一般同时使用，如图 23－55 所示。快速恢复机制是这样的：

- 发送端连续收到三个重复 ACK 确认报文后将 ssthresh 设置为当前 cwnd 大小的一半。
- 接下来并不执行慢启动，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。

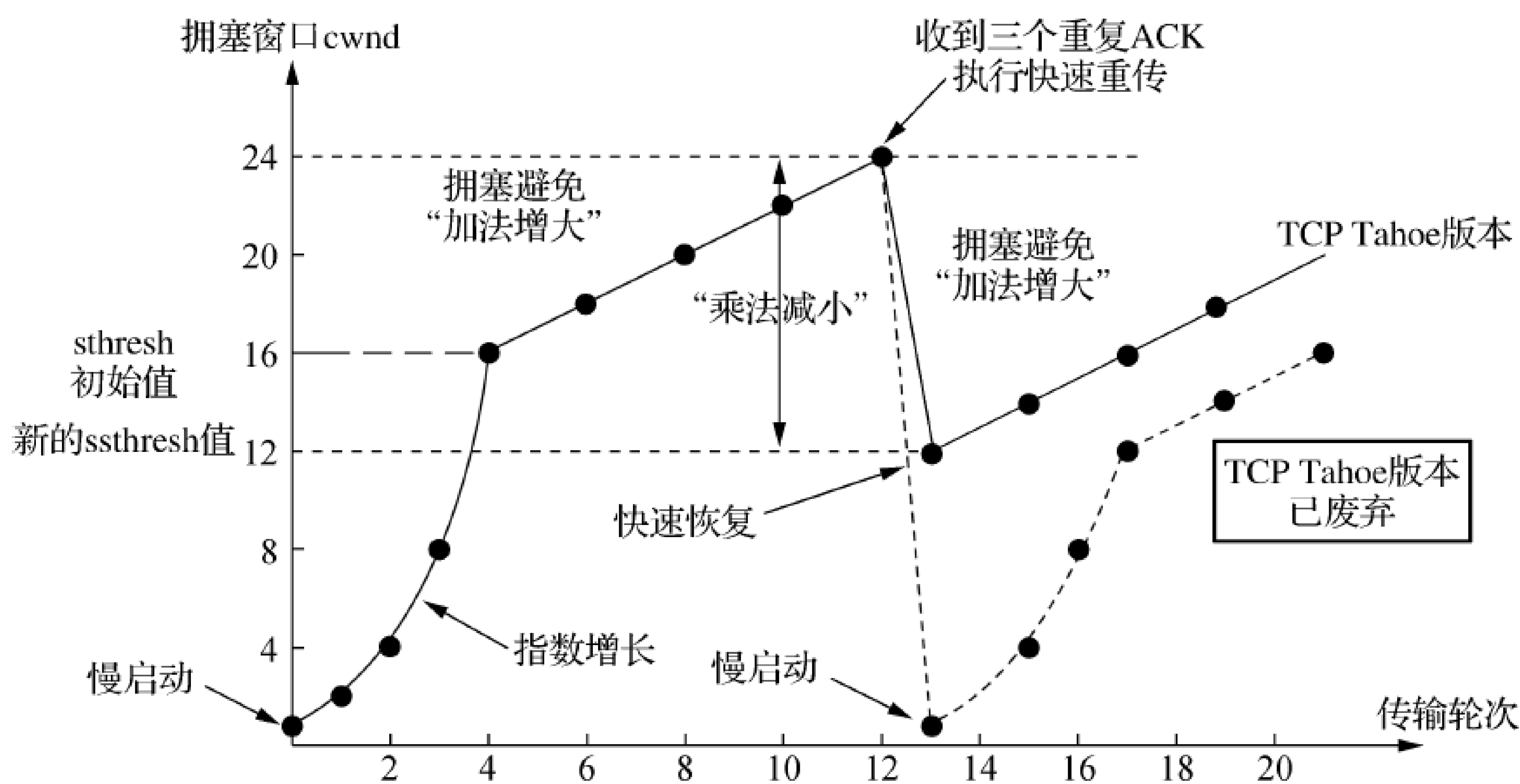


图 23－55　TCP 中的快速重传与快速恢复过程

需要注意的是，快速重传机制依赖于接收端连续发送三个重复的 ACK 确认报文，不过这三个 ACK 并非代表只丢了一个报文段，也有可能是多个报文段，但快速重传算法只会重传一个报文段，剩下的只能有赖于 RTO 超时了，这也是快速重传算法的一个弊端。

TCP 的 New Reno 版本和 SACK 机制可以解决上述问题。

3）拥塞控制状态机

TCP 有自己的状态机,拥塞控制同样也有自己的状态机,发送端 TCP 协议通过状态机来决定下一步的动作(是增大 cwnd 还是缩小 cwnd 抑或保持不变)。拥塞控制状态机如图 23－56 所示。

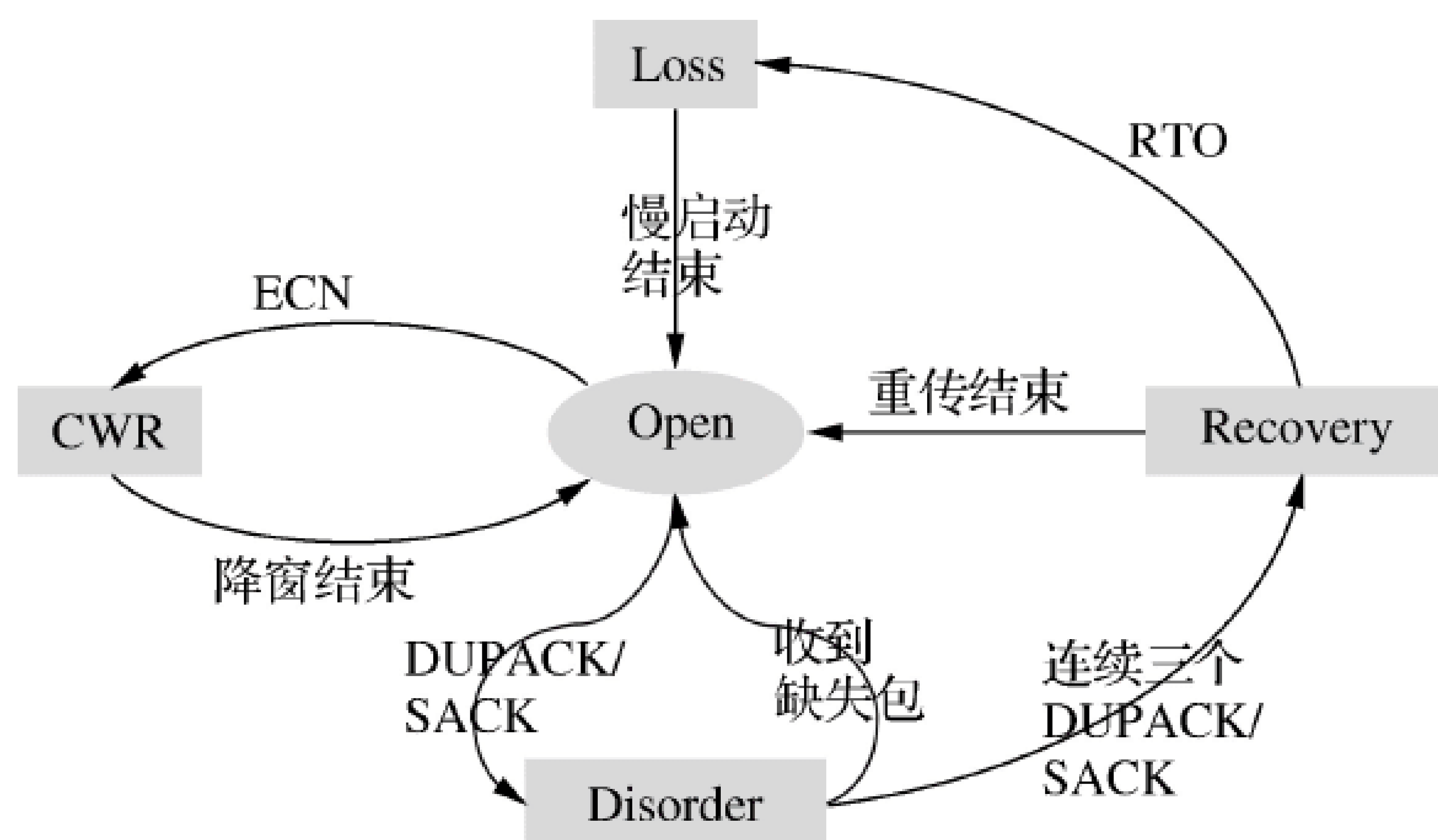


图 23－56　TCP 的拥塞控制状态机

拥塞控制状态机有五大状态：

- **Open**：这是拥塞控制状态机的默认状态。发送端接收到 ACK 确认报文后会根据 cwnd 大于还是小于 ssthresh 来决定接下来执行慢启动还是拥塞避免。
- **Disorder**：发送端在收到重复确认报文(DUPACK)或者选择性确认报文(SACK)时会认为发生了分组失序,网络中负载已达极限,此时转变为 Disorder 状态。在这种状态下发送端只有在一个已发出的包离开了网络后(判断标准为收到该包的 ACK 确认报文)才会发出新的包。
- **CWR**：CWR 即 Congestion Window Reduced(拥塞窗口减少),发送端在拥塞发生时并不是立即减少拥塞窗口,而是每当收到两个 ACK 确认报文就减少一个单位的 MSS,直到 cwnd 的大小减半为止。cwnd 减小且网络中没有重传包时的状态就是 CWR 状态。
- **Recovery**：发送端接收到三个 DUPACK 后进入该状态。在此状态下,接收端每收到两个 ACK 确认报文就减少一个单位的 MSS,直到 cwnd 等于 ssthresh。此时的 ssthresh 等于刚进入 Recovery 状态时的一半大小。发送端会从该状态逐渐恢复成 Open 状态,但如果重传超时则会进入 Loss 状态。
- **Loss**：当一个 RTO 到期后,发送端进入 Loss 状态,并将其所有发送的数据标定为丢失,cwnd 被初始化为 1(1 个 MSS),开始慢启动过程。

4）随机早期检测

随机早期检测(Random Early Detection,RED)是路由器为了避免发生全局同步现象而采取的一种策略,其原理是：

- 路由器分组缓冲一般采用队列的方式,RED 机制在路由器中维护两个水位值参数,即队列长度最小阈 min 和最大阈 max,分别代表了水位值下边界和上边界的两个阈值位置。
- 分组到达路由器后 RED 计算队列的平均长度：
    - 如果平均长度小于 min,则分组正常入队；

- 如果平均长度介于 min 和 max 之间,则按照某个概率 p 丢弃分组;
- 如果平均长度大于 max,则丢弃新分组。

这里最关键的就是以概率 p 丢弃分组,使拥塞只发生在个别的 TCP 连接上,从而避免全局同步现象的产生(如图 23－57 所示)。拥塞发生与避免不是端和端之间的问题,而是端与链路之间整体的问题,因此在解决了接收端与发送端的拥塞控制后,也要着手优化链路(路由器等)的拥塞控制。

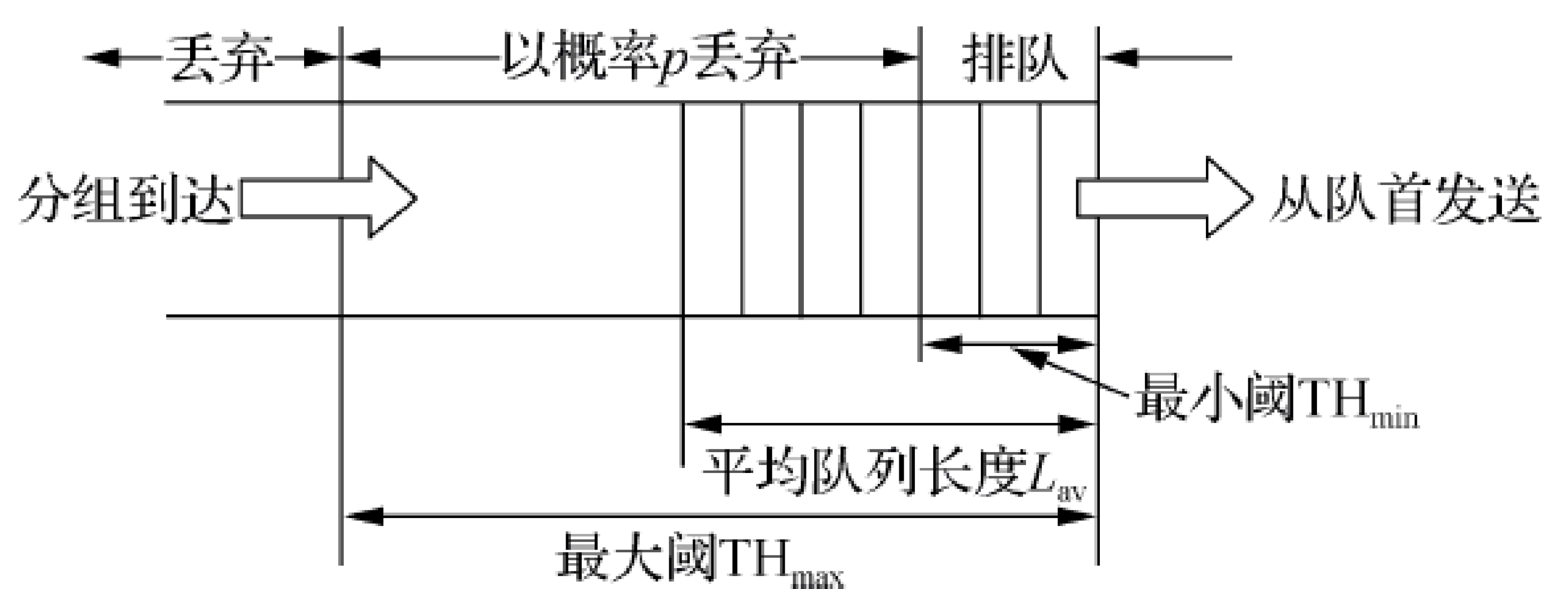


图 23－57　随机早期检测过程

5）TCP 的拥塞控制算法

TCP 的拥塞控制算法也代表了 TCP 的版本,也就是说 TCP 的演进本质上是拥塞控制算法的演进。下面我们先来介绍一下主要的 TCP 版本。在 TCP 的发展过程中,大致有 4 种不同思路的拥塞控制算法:

（1）**基于丢包的拥塞控制算法**:将丢包视为拥塞发生,因此采取探路和逐渐增大拥塞窗口的办法,例如 Reno、Cubic 等算法。

（2）**基于时延的拥塞控制算法**:将时延增加视为拥塞发生,因此时延增加时增大拥塞窗口,时延减少时缩小拥塞窗口,例如 Vegas、FastTCP 等算法。

（3）**基于链路容量的拥塞控制算法**:实时测量网络带宽和时延,认为网络上报文总量大于带宽时延乘积时就是拥塞发生,例如 BBR 算法。

（4）**基于学习的拥塞控制算法**:与上面几种思路不同,这种算法借助评价函数并基于训练数据,使用机器学习的方法形成一个控制策略算法,例如 Remy 算法。

下面我们列举几种典型的拥塞控制算法,或者说是 TCP 的版本。

➢ **TCP Vegas**:是于 1995 年提出的利用 RTT 来判断拥塞和丢包的 TCP 版本,该版本在长肥网络中不常用。在该版本中,当 RTT 变大时,有理由认为链路发生了拥塞,此时减小 cwnd;反之,如果 RTT 变小,则认为拥塞情况发生了好转,此时增大 cwnd。当收到一个重复的 ACK 确认报文时会比较从发送这个数据报文段到当前时刻的间隔,如果大于 RTT 则认为发生了丢包,无需等待超时或者收到三个重复 ACK 确认报文就直接重传该报文段。不过在与其他控制算法共存的情况下,由于基于丢包的算法会尝试填满网络中的缓冲区继而导致 RTT 增大,使得 Vegas 算法减小了 cwnd,最终导致传输速率变得越来越慢。

➢ **TCP Tahoe**:是 TCP 的早期版本,在这个版本中规定了慢启动、拥塞避免和快速重传三个阶段。当拥塞发生时,cwnd 又从 1 开始慢启动。在这种方式下传输效率是个大问题。

➢ **TCP Reno**:是 TCP 的较新版本,在这个版本中增加了快速恢复阶段,也就是将发生拥塞时的慢启动直接改为了拥塞避免,这样 cwnd 也不会大起大落"硬着陆"。Reno 版

本适用于低延时、低带宽的网络。

- **TCP New Reno**：是较 TCP Reno 更新的版本，解决了前一版本中问题和弊端，无需等待 RTO 超时就可以直接持续发送所有丢失的报文段（大致为一个 RTT 周期重新发送一个丢失的段）。
- **TCP SACK（Selective Acknowledgment，选择性确认）**：是最新的 TCP 版本，是对 TCP New Reno 的扩展，支持只对丢失的报文段进行确认（采用 TCP 的扩展字段实现），使得发送端清楚地知道哪些数据丢了而应该重发。
- **TCP Cubic**：Cubic 是 Linux 内核 2.6 版本之后默认的 TCP 拥塞控制算法。该算法使用一个立方（cubic）函数作为 cwnd 的增长函数，cwnd 的增长不再与 RTT 有关，而仅仅取决上次发生拥塞时的最大窗口和距离上次发生拥塞的时间间隔。该算法的优点在于只要没有出现丢包，就不会主动降低自己的发送速度，可以最大限度地利用网络剩余带宽，提高吞吐量，在高带宽、低丢包率的网络中可以发挥较好的性能。

除此之外，还有一些优化的拥塞控制算法也都是基于上述算法做了有限的改进，例如 HSTCP（High Speed TCP）、TCP BIC、TCP WestWood 等算法。

6）BBR 算法

我们在前文说过，TCP 有两大控制机制：拥塞控制和流量控制，这两者本质上是相辅相成的。拥塞控制是对发送端的下行流量（发送方向的流量）进行控制，规定了下一个时刻可以发送多少，但并不意味着仅此而已，这里面还包括了一个怎么发的问题。打个比方，我们有 1 000 包数据需要在 1 秒（1 000 毫秒）内发出，那我们可以在第 1 毫秒发出 1 000 包而在后面 999 毫秒不作为，也可以每 1 毫秒发出 1 包。显而易见，后者对于流量的控制更好更合理，更不容易产生拥塞和路由器内排队现象。遗憾的是，TCP 的拥塞控制机制基本上没有对怎么发包做出规定。

2016 年，Google 发布了一种新的 TCP 优化传输算法 BBR（Bottleneck Bandwidth and Round-trip propagation time，瓶颈带宽和往返传播时间），并集成到了 Linux 4.9 内核中。BBR 算法采用了与之前几种拥塞控制策略不同的方式，即采用了主动探测带宽与 RTT 的方式来进行拥塞控制，并基于此对流量控制也做了规定。

传统的 TCP 拥塞控制算法存在以下不合理之处：

- **首先是拥塞的判断机制**。传统拥塞控制算法认为发生丢包就是产生了拥塞，但这个结论不全对。因为在一般的局域网中，由于链路错误等原因造成的丢包是可以忽略不计的，但在长途的广域网中，由于各种原因造成的丢包就很频繁地发生了，特别是经过无线传输阶段，误码率非常高，这部分原因造成的丢包是不能忽略的，而这种丢包就不是因拥塞而产生的了。因此传统的拥塞控制算法在应对这种长肥网络的时候过于“一刀切”。
- **其次是缓冲区膨胀问题**。网络中的缓冲区用于吸收流量波动。拥塞控制算法倾向于填满整个缓冲区，但当缓冲区较大而又遇上网络拥塞的情况时，需要很长时间才能清

空缓冲区中的数据，这会造成网络延时。也就是说，发送端估计的发送窗口总是略大于真实链路容积，这种情况被称为“缓冲区膨胀”。膨胀往往会恶性循环地加剧吞吐量的下降，继而产生丢包。

➢ **最后是在长肥网络中的窗口计算问题**。广域网的发送窗口大小一般会比局域网的大 1 ~2 个数量级，但错误丢包率需要至少低 2 个以上的数量级才能正常工作，这是因为广域网的 BDP 高于局域网好几个数量级，也就是发送窗口大小大了好几个数量级，但错误丢包率需要与发送窗口大小的平方成反比，因此在长肥的广域网中只会收敛到一个很小的发送窗口（长肥网络的错误丢包率是比较高的）。此时虽然通信两端和中间链路都有空间可用，但是发送速度很慢，甚至发送到一半就没速度了。

为了解决这些不合理的问题，BBR 算法引入了新的机制和算法。

（1）新的瞬时带宽算法

BBR 算法会跟踪目前为止最大的瞬时带宽，这个带宽将作为 BBR 算法的基础数据（该带宽是个标量数据，因此不必纠结于其含义）。瞬时带宽的计算只需要两个数据：应答了多少数据，记为 delivered；应答 delivered 这么多的数据所用的时间，记为 interval_us。BBR 定义的瞬时带宽如下：

$$瞬时带宽 = delivered/interval_us$$

每隔 10 秒探测一次瞬时带宽，这个带宽数据从计算的来源上来讲没有什么意义，BBR 算法也不关心它的含义。同时，这里的应答既包括 ACK 也包括 SACK，BBR 算法认为，即使是被 SACK，也代表链路上已经清空了多少网络数据包。

从图 23 – 58 所示的例子中可以看出，BBR 算法不关心数据包是否按序确认，也不关心是 ACK 还是 SACK，而是只要收到确认包就算数，这也代表了 BBR 算法真正关心的是链路上网络包被清空的数量，而不是接收端已经成功接收了多少包。好像很有道理的样子。

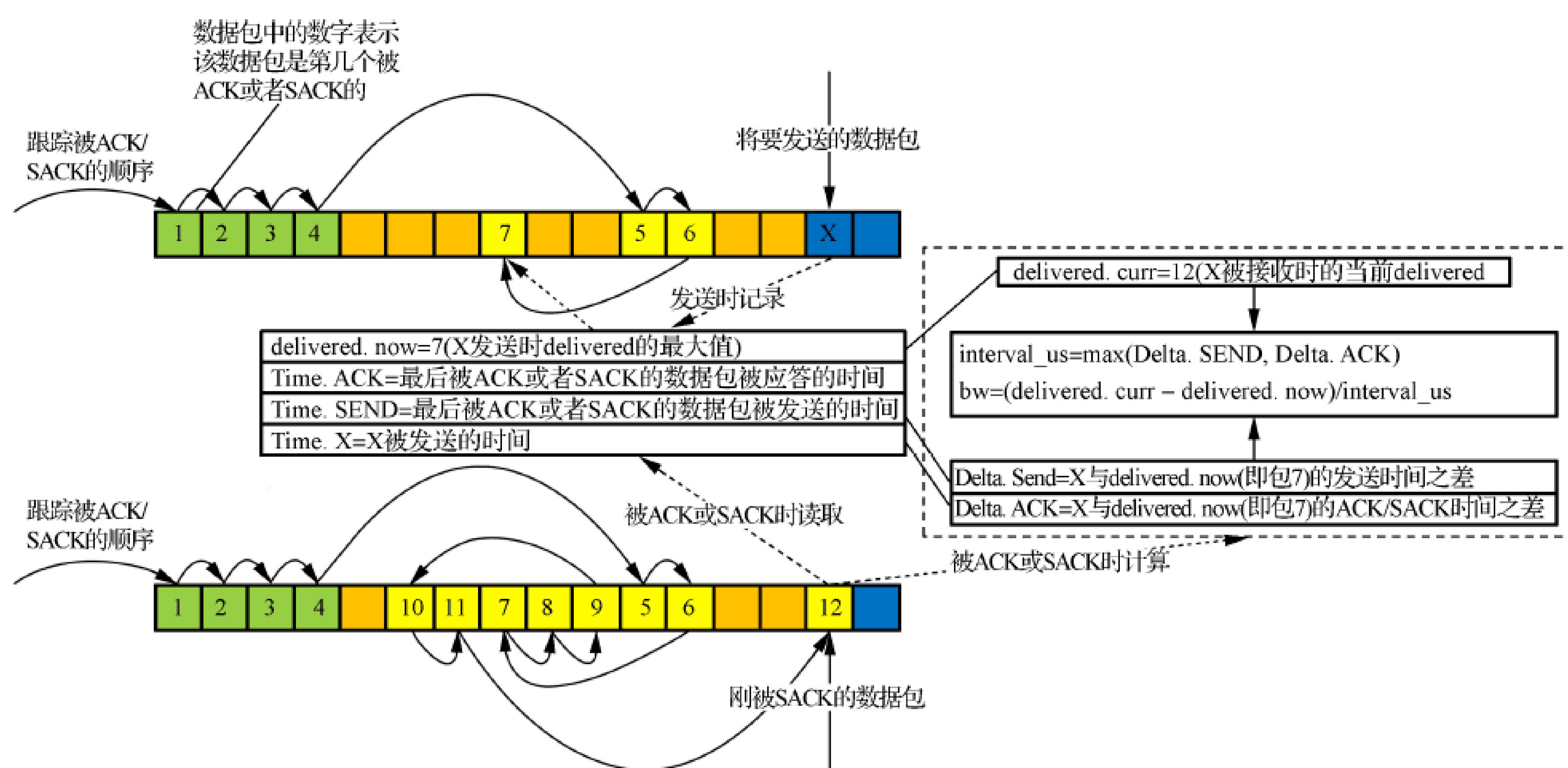


图 23 – 58　瞬时带宽的计算（例子中共有 12 –7 =5 个数据包被 ACK/SACK）

(2) 最小 RTT 跟踪

BBR 算法采用主动探测机制获取 RTT,这个 RTT 是目前为止的最小 RTT。

(3) pipe 状态机

根据互联网上的拥塞行为,BBR 算法不再采用 TCP 的拥塞控制状态机了,而是另外提出了 4 种状态:STARTUP(起始状态)、DRAIN(清空状态)、PROBE_BW(瞬时带宽探查状态)和 PROBE_RTT(RTT 探查状态),如图 23 - 59 所示。

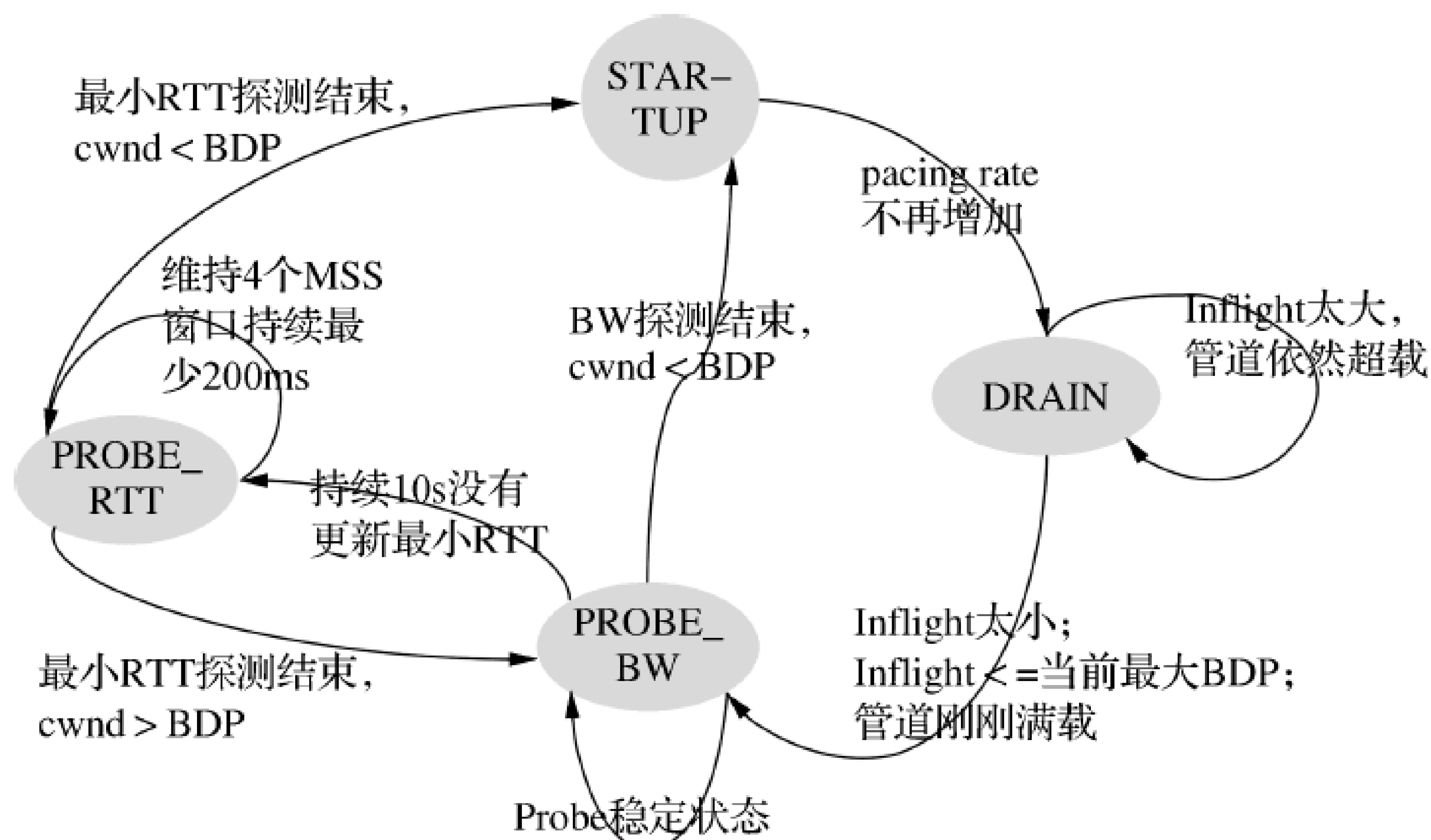


图 23 - 59　BBR 算法的状态机

上述 4 个状态中,STARTUP 是最初的起始状态,此时以带宽的 bbr_high_gain(增益系数)为倍数计算发送速率 pacing rate 与拥塞窗口 cwnd。当发送速率不再增加时进入 DRAIN 状态。如果拥塞窗口(Inflight)不大于当前最大的 BDP(带宽时延乘积),证明链路刚刚满载,此时进入稳定状态 PROBE_BW。在这个状态下 BBR 算法会定时探测 RTT 和链路瞬时带宽 bw,当拥塞窗口小于 BDP 时进入 STARTUP 状态,并再次试图增加发送速率。这是一个周而复始的过程。

- **增益系数(bbr_high_gain)**:增益系数大于 1,代表接下来可以比计算出的 bw 值更快更多地发送数据;增益系数小于 1 则正好相反,pacing rate 与 cwnd 都要根据 bw(链路瞬时带宽)计算的结果下调以达到降速目的;如果增益系数正好为 1,则按照当前计算的 bw 值发送。
- **稳定状态**:在该状态下计算 pacing rate 与 cwnd 的时候并不像初始状态中采用一个很大的增益系数去扩张或收缩它们,而是用一个比较中庸平和的增益系数(例如 1、3/4、5/4)去扩张或收缩,就像轻踩油门与刹车一样,以达到一种轻微浮动式的波动状态。稳定状态极力避免熔断式"硬着陆"。

在传统的 TCP 拥塞控制算法中,cwnd 总是测不准,因为传统算法倾向于同时测量代表发送容量的 BDP,也就是同时测量 bw 与 RTT。但是 bw 最大也就意味着链路上的缓冲区中的数据"膨胀"了,RTT 就是最大;反过来说,RTT 最小就意味着链路缓冲区为空,但带宽肯定不是最大,因此这两个结果总是测不准,乘积总是上下波动,再加上一些激进的拥塞避免和

快速恢复算法,这个波动就比较剧烈了。

BBR不再同时测量bw与RTT了,而是分开交替测量(如图23－59中的状态机所示),并且不以外部条件为判断依据,心无旁骛地以自己的测量结果(bw和RTT)为准,并且将测量的周期拉长,以一段时间内采样的最大bw和最小RTT作为估计值。这意味着BBR算法相信网络是长期向好的。

(4) pcaing rate(发送速率)与cwnd(拥塞窗口)的算法

这两个值定义了数据怎样发出去、以多大的时间间隔发出去、发送多少数据。这是为了避免"突发"情况而造成的路由器排队、RTT抖动等情况而设置的。有的时候接收端为了节约带宽把多个ACK确认报文封装为一个包返回,使发送端一股脑地发出好多包;还有些时候上层应用传来的数据量突然增大,也会导致一股脑发大量数据的突发情况发生。

pacing rate的计算非常简单,就是使用时间窗口(默认10轮采样周期)内最大的链路瞬时带宽bw乘以当前的增益系数。

cwnd也描述了管道的容量,即BDP的值。因此BBR算法首先使用最小的RTT(代表了曾经达到的最佳RTT)计算BDP,既然以前达到过,那现在和将来也有可能再次达到。计算出BDP后乘以增益系数就是cwnd的值。

下面介绍BBR算法的工作过程。

① **慢启动阶段**:BBR算法启动时也会有一个类似慢启动的阶段,即乘数级增大发送速率。在这个阶段,中间链路的缓冲区未被占用,因此RTT很小,并作为初始的采样估计值,此时的瞬时带宽bw作为带宽的采样估计值。

② **拥塞避免(清空)阶段**:当BBR算法收到确认包后如果发现有效带宽不再增长了就进入了拥塞避免阶段。在这个阶段只要丢包率不是太高,BBR算法就基本视而不见。当发送速率增大到开始占用中间链路缓冲区的时候,有效带宽就不再增长了,这样保证了缓冲区不会被填满。但此时已经开始占用缓冲区了(实测会多占用2倍带宽延时乘积的缓冲空间)。这时要把多占用的缓冲空间清空,因此会乘数级降低发送速率(从STARTUP状态转入DRAIN状态)。

③ **稳定状态运行阶段**:这时清空阶段结束,BBR算法交替探测带宽和RTT(实际上稳定状态的绝大多数时间处于带宽探测阶段),这时处于正反馈场景——定期增大发包速率,如果收到确认的速度也提高了,就进一步增大发送速率。

从上述信息中可以看出,BBR是个不计较丢包的拥塞控制算法,更没有把传统TCP的拥塞状态(Loss、Recovery)当回事。BBR算法只根据瞬时带宽bw来调整代表发送量的cwnd,并根据bw计算发送速率。BBR算法真正在意的是在一段时间内没有达到RTT采样的最小值或更小的值,这证明链路没有在变好可能正在变差,也许就意味着发生了拥塞。

BBR算法不断地基于当前带宽以及当前的增益系数计算调整pacing rate及cwnd,以此结果作为拥塞控制算法的输出。每收到一个ACK确认包,都会计算瞬时带宽,然后将结果反馈给BBR的pipe状态机,并不断地调节增益系数,这就是BBR算法的核心工作机制,如

图 23-60 所示。因此 BBR 算法是一个典型的封闭反馈系统,与 TCP 当前处于什么拥塞状态完全无关。这一切机制都瞄准了尽最大努力充分高效地占用带宽和降低链路缓冲区占有率从而减少延迟的理念。

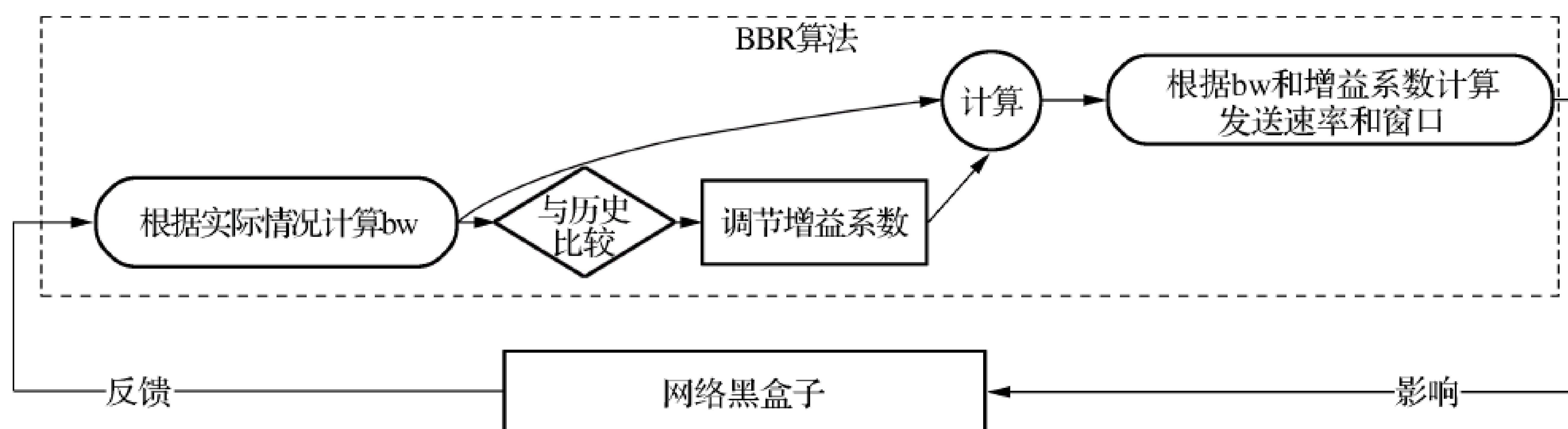


图 23-60　BBR 算法的工作机制

BBR 算法也不再使用"加法增长,乘法减小"的方式来维护发送窗口大小,而是分别估计极大带宽和极小延迟,把它们的乘积作为发送窗口的大小。

目前,BBR 算法在长肥网络中已经越来越被广泛应用了,在前文介绍的 QUIC 协议中就采用了 BBR 算法改善 TCP 的拥塞控制机制。

23.2.6　TCP 加速技术

所谓 TCP 加速就是通过改进发送或拥塞控制的策略来提升 TCP 的发送效率和发送速率,本质上就是通过判断拥塞以调整发送速率。在长肥的广域网中 TCP 加速特别重要,可以分为双边加速和单边加速两种。

- **双边加速**:要在 TCP 的接收端和发送端都进行加速升级,因此也可以笼统地看作一种私有协议。既然是私有协议,普通版本的 TCP 就不支持它了,这在一定程度上限制了 TCP 加速技术的普及。双边加速的算法和技术很多,特别是基于拥塞判断及拥塞控制的算法,在此我们不详细叙述。
- **单边加速**:这是一种只需要在一端(一般是发送端)进行升级和优化的加速技术。采用单边加速的 TCP 在提高了发送速率的同时,却不需要通信对端做任何升级,具有更好的兼容性。

以拥塞的判断及处理方式来区分,到目前为止,TCP 加速技术已经历了两代演进:

- **基于丢包的拥塞判断及处理(Loss-based)**:以丢包来判断拥塞并调整传输速率。
- **基于延迟的拥塞判断及处理(Delay-based)**:以往返延迟 RTT 的变化来判断拥塞并调整传输速率。

不论是基于丢包还是基于延迟,加速技术都采用静态算法,即基于对流量模型的假设前提而采用固定的拥塞判断及恢复机制。但网络的流量特征越来越复杂、越来越动态并难以预测,因此基于丢包和基于延迟的加速技术常常只在假设前提成立的特定网络场景下有效,并且随着网络路径特征的变化,加速效果也起伏不定,有时甚至出现反效果。

随着机器学习技术的发展，基于学习的（Learning-based）拥塞判断及处理的优势逐渐显现出来。与上述两代技术不同，基于学习的加速技术通过对路径上传输历史的学习来动态地判断拥塞并调整传输速率。本节我们讨论两种基于学习的单边加速技术。

1）ZetaTCP 加速技术

ZetaTCP 是华夏创新（AppEx）在 2006 年研发的 TCP 单边加速引擎，支持智能流控算法，并且兼容传统的 TCP 流控机制，使用 ZetaTCP 算法时无需对 TCP 头做任何修改。ZetaTCP 具有以下加速特点：

（1）引入了精确的丢包判断和预测算法

- 传统的 TCP 流控机制经常误判丢包，例如将一些重传行为误判为丢包，从而造成了带宽和时间的浪费。
- ZetaTCP 改进了丢包判断的机制，采用网络路径特征自学习的动态算法，基于每一个 TCP 连接实时观察并分析网络特征，根据学习到的网络特征随时调整算法来更准确地判断拥塞程度并及时地判断是否丢包，从而更恰当地进行拥塞处理并更快速地进行丢包恢复。

（2）定时侦测链路带宽并根据侦测结果调整发送速率

- 传统的 TCP 滑动窗口机制也经常误判路径上的带宽容量，或者增得太急或者减得太猛，出现这两种极端震荡状况时也往往伴随着大量丢包，导致了带宽利用率下降。
- ZetaTCP 根据主动侦测的路径带宽动态调整数据发送量。

（3）实时监测和学习接收端的传输行为

- 传统的 TCP 是基于被动确认方式的，这种方式需要发送端等到接收端返回 ACK 确认报文时才能研判丢包并估算路径带宽。
- ZetaTCP 根据发送模式智能地反馈和引导对端判断丢包以及精确计算路径带宽。

实践表明，在延迟较大、丢包率较高的网络中，ZetaTCP 相比基于丢包和基于延迟的两种加速方式表现出明显的优势。

2）mmTrixTCP 加速技术

mmTrixTCP 也是一种基于传输历史学习的单边 TCP 加速技术，在传统拥塞控制算法的基础上增加了下列加速机制：

（1）使用科学的拥塞反馈信号

作为新一代 TCP 加速技术，mmTrixTCP 同时考虑丢包和延迟的变化，并且引入了 TCP 连接路径网络特征自学习的动态算法来进一步提升拥塞判断的准确性和及时性。mmTrixTCP 通过动态学习分析的方式推导出特定 TCP 链路上反映拥塞的前兆信号，过滤掉并非由拥塞导致的丢包和延迟变化，从而使拥塞判断更及时、更准确。

（2）建立更好的数学模型

可以减少慢启动和线性增加所做的逐步探测网络带宽的过程，从而更快地达到网络最大利用率。mmTrixTCP 根据拥塞反馈信号动态计算一个网络的均衡点（在这个点实现了网

络利用率的最大化,并且能尽量保证各方的公平性)。离均衡点越远,拥塞窗口调整越快,否则越慢。

(3) 使用平滑的突发控制方法

mmTrixTCP 突发控制部分通过一种类流方式来追踪网络的可用带宽,以达到平滑数据包发送的目的。突发控制在大带宽时延乘积网络中的作用尤为重要,因为在此类网络中有时受网络和末端主机的影响,造成的突发通信量可能会非常大。

比如,一个确认消息能够确认上千个数据包,这样会使发送窗口突然变得非常大而造成“突发”,有时也会造成发送端的 CPU 被长期占用以处理输入的待发送数据包。此时允许输出的数据包堆积在设备输出队列中,当 CPU 可用时,就可能产生巨大的脉冲,这种脉冲会产生长队列,并会增加数据包大量丢失的可能性。

mmTrixTCP 使用两种突发控制机制:一是对单个数据包补充自同步,二是使用连续小脉冲使窗口增加更加平滑。

(4) 使用基于预测的丢包判断

标准 TCP 协议栈通过两种手段判断丢包:一是接收端连续重复 ACK 包的数量,二是 ACK 超时。当有较多丢包时,往往要靠 ACK 超时来判读超时并引发重传。大带宽时延乘积网络的丢包是常态,一个连接上有上千个数据包同时丢失是常有的事。因此标准 TCP 经常要靠超时来重传,这会使传输被长时间阻塞。这是影响标准 TCP 效率的又一主要原因。

mmTrixTCP 的丢包判断除了采用上述标准 TCP 的两种手段外,还引入了 TCP 连接路径网络特征自学习的动态算法来尽快地预测丢包。预测算法考虑的网络特征因素和 mmTrixTCP 拥塞判断的自学习算法类似。通过传输历史的跟踪学习,mmTrixTCP 丢包判断算法随机地对已发出去但尚未被对方 ACK 的数据包给出一个已丢失概率,概率值会随着传输的进行不断调整。当概率达到一个很高的阈值时,算法认为当前发出的包已丢,将立即启动重传。

23.3 应用层协议

在前面两节中,我们分别介绍了位于 TCP/IP 模型第三层的网络层协议和第四层的传输层协议。本节将介绍位于 TCP/IP 模型最上层的应用层协议,特别是聚焦于视联网(VoIP)和物联网(IoT)领域的应用层协议。更确切地说应该是协议栈,因为包括会话层和表示层在内的跨层协议也会被纳入,这也是整个计算机通信协议中比较细分的领域协议。这些协议更加贴近应用程序,更贴近行业,更需要与应用相匹配。

23.3.1 视联网协议栈

在此,我们整理了常见的视联网协议列表,包括视频监控、视频会议、视频封装等多个细分场景的协议(如图 23-61 所示),有的协议只能在某个细分领域出现,而有的协议则可以

跨越多个视频场景。这些协议按照功能可以分为协商类、封装类、编码类和传输类等，这些协议的组合又可能构成更上层的视频细分行业的协议（例如 GB/T 28181 就是基于 SIP、SDP、RTP 等协议构成的视频互联互通应用协议），因此我们称之为视联网协议栈也是很贴切的。

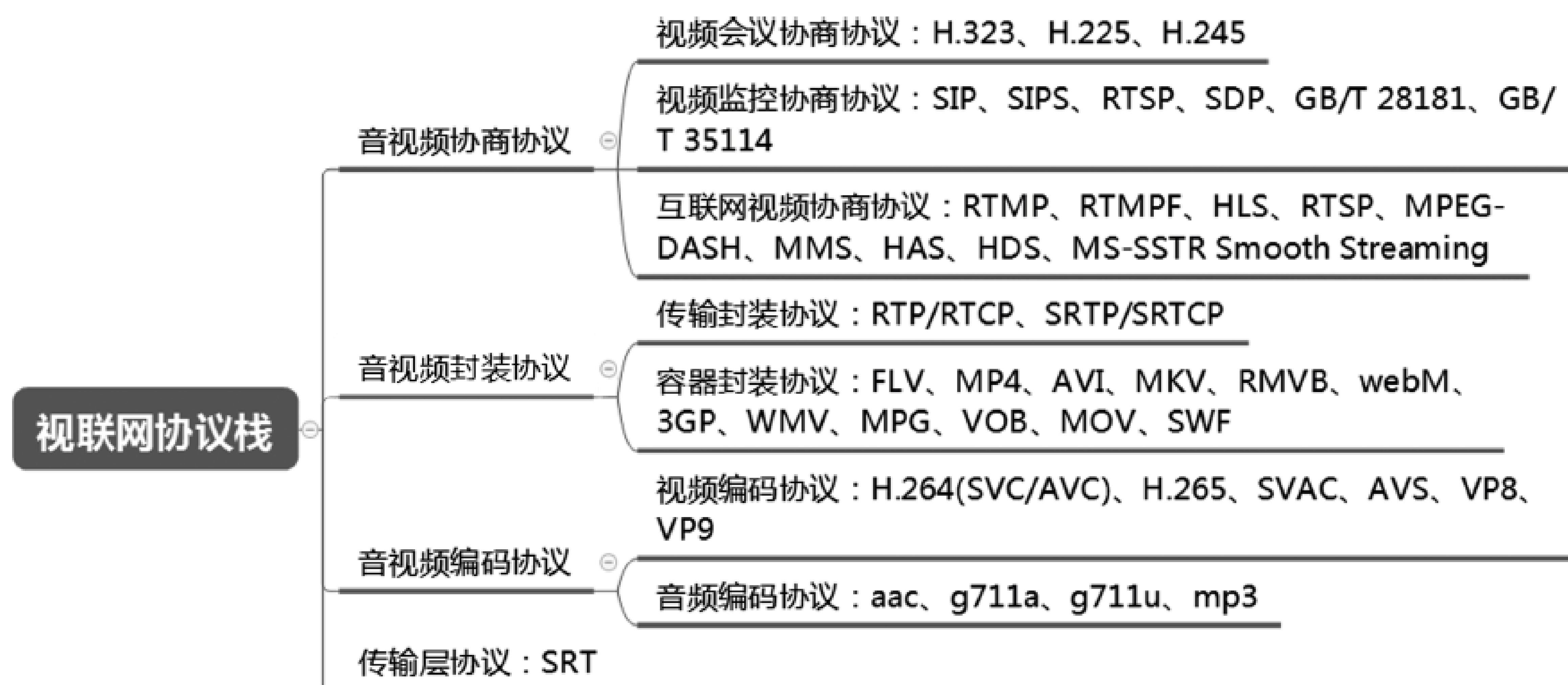


图 23－61　常见视联网协议列表

不过本节对于视联网协议栈只是泛泛而谈，不会“贪大求全”地什么都介绍，也不会介绍得太细。这是因为大部分协议的详细资料在网上多如牛毛，而且这些协议大多也都属于“博学强记”类的，是对规则规范的遵守。

对于某些应用层协议，掌握其功能、数据报文格式和工作流程就可以了。例如流媒体领域的协议 RTSP 与 RTMP，其握手与传输过程可以通过抓包比对着看，这样更有益于加深理解，如图 23－62 所示。

Source	Destination	Protocol	length	New Column
10.45.148.209	10.45.149.196	RTSP	179	OPTIONS rtsp://10.45.149.196:554/000100 RTSP/1.0
10.45.149.196	10.45.148.209	RTSP	206	Reply: RTSP/1.0 200 OK
10.45.148.209	10.45.149.196	RTSP	205	DESCRIBE rtsp://10.45.149.196:554/000100 RTSP/1.0
10.45.149.196	10.45.148.209	RTSP/SDF	941	Reply: RTSP/1.0 200 OK
10.45.148.209	10.45.149.196	RTSP	233	SETUP rtsp://10.45.149.196/000100/trackID=1 RTSP/1.0
10.45.149.196	10.45.148.209	RTSP	257	Reply: RTSP/1.0 200 OK
10.45.148.209	10.45.149.196	RTSP	252	SETUP rtsp://10.45.149.196/000100/trackID=2 RTSP/1.0
10.45.149.196	10.45.148.209	RTSP	257	Reply: RTSP/1.0 200 OK
10.45.148.209	10.45.149.196	RTSP	211	PLAY rtsp://10.45.149.196/000100/ RTSP/1.0
10.45.149.196	10.45.148.209	RTSP	426	Reply: RTSP/1.0 200 OK

<table>
<tr><th>Source</th><th>Destination</th><th>Protocol</th><th>length</th><th>New Column</th></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>131</td><td>Handshake C0+C1</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>130</td><td>Handshake C2</td></tr>
<tr><td>10.45.157.252</td><td>10.45.148.49</td><td>RTMP</td><td>130</td><td>Handshake S0+S1+S2</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>158</td><td>connect('hls')</td></tr>
<tr><td>10.45.157.252</td><td>10.45.148.49</td><td>RTMP</td><td>70</td><td>Window Acknowledgement Size 5000000</td></tr>
<tr><td>10.45.157.252</td><td>10.45.148.49</td><td>RTMP</td><td>289</td><td>Set Peer Bandwidth 5000000,Dynamic|Set Chunk Size 4096|_result</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>99</td><td>releaseStream('mystream')</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>95</td><td>FCPublish('mystream')</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>87</td><td>createStream()</td></tr>
<tr><td>10.45.157.252</td><td>10.45.148.49</td><td>RTMP</td><td>95</td><td>_result()</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>104</td><td>publish('mystream')</td></tr>
<tr><td>10.45.157.252</td><td>10.45.148.49</td><td>RTMP</td><td>171</td><td>AMF0 Command</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>70</td><td>Set Chunk Size</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>197</td><td>AMF0 Data</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>98</td><td>Video Data</td></tr>
<tr><td>10.45.148.49</td><td>10.45.157.252</td><td>RTMP</td><td>1415</td><td>Video Data|AMF0 Command</td></tr>
</table>

图 23－62　RTSP 与 RTMP 协议的握手交互流程对比

对于另外一些协议，其状态机、对话(dialog)、会话(session)、事务(transaction)、定时器等则更为核心和底层。例如视频监控和视频会议广泛使用的SIP协议，其核心就是状态机机制、超时机制和事务机制等。学习这样的协议，一定要深刻理解其内部机理。

下面我们会根据上述思想介绍基于传输层UDT的流媒体"专属"协议SRT、音视频封装容器协议和RTP协议的加密机制(SRTP)、视频会议协议簇以及视频安防行业相关协议。

23.3.1.1 SRT协议

SRT(Secure Reliable Transport，安全可靠的传输)是一种开源的音视频传输协议，这包含了两层含义：① 这是传输层协议；② 这是音视频"专属"协议。SRT是由Haivision公司和Wowza共同创建的SRT联盟所发起的互联网传输协议，这也就代表了第三层含义：这是一种适用于互联网的协议(当然也适合局域网，但在局域网下可以选择其他更好的传输协议)。

SRT是一种基于UDT的协议并且延续了UDT的拥塞控制机制，其基本出发点是为了解决UDT的传输延迟高的问题，能够处理长时间的网络延迟。SRT也具有很高的适配性，由于SRT与本身的负载无关，只要能够使用UDP传输的数据报文都可以使用SRT传输，因此可以支持多种流传输，并支持多个并发流。除此之外SRT协议还具有以下特性：

- **穿透特性**：与UDT一样，SRT协议支持防火墙和私网穿透。
- **UDP兼容性**：与UDT一样，任何能够兼容UDP的数据都可以承载于SRT之上。
- **时间同步特性**：每个SRT数据包都具备由发送端设置的高精度时间戳，这个时间戳可以被接收端解码。传输过程中上下游之间的延迟可以通过时间戳精确掌握。
- **直接建立连接**：可以在发送端和接收端之间建立连接而无需中间的中转服务器，消除了中心的瓶颈进而减少了延迟。
- **使用便捷**：SRT协议具有免费开源的基于API的代码库，并具有自己的开源社区。

下面我们来看SRT协议的关键工作流程。

1. 握手和信息交换

SRT提供了三种不同的握手协商模式，即调用模式、侦听器模式和汇聚模式。

- 调用模式有点类似TCP的客户端，即一个SRT端点向另一个已知IP地址和UDP端口号的端点发起连接请求。
- 侦听器模式有点类似TCP的服务端，即本端监视传入的连接请求，等待来自调用模式设备的连接。
- 汇聚模式比较类似UDT的会合连接模式，两个SRT端点都同时充当调用者和侦听者，也都由本端向对端发起连接请求，以便穿透私网或者防火墙。

SRT协议每次握手时都要进行双向确认，通过安全Cookie对之前已经建立的对端标识和密码进行验证。握手完成后，双方交换各自的功能和配置，以便让对端明了自己的能力集。如果要对音视频数据进行解密，也可以在这期间交换加密密钥。

2. 使用ARQ机制传输数据

在ARQ(Automatic Repeat-Request，自动重传请求)机制下发送端可以根据接收端的请

求重新发送传输过程中丢失的包。图 23－63 对比了在未纠错、前向纠错(FEC)和 ARQ 三种机制下的传输方式,可以看出,FEC 机制会在链路质量和状态良好的情况下增加传输内容,降低了传输的有效性,并进而导致额外的延时。

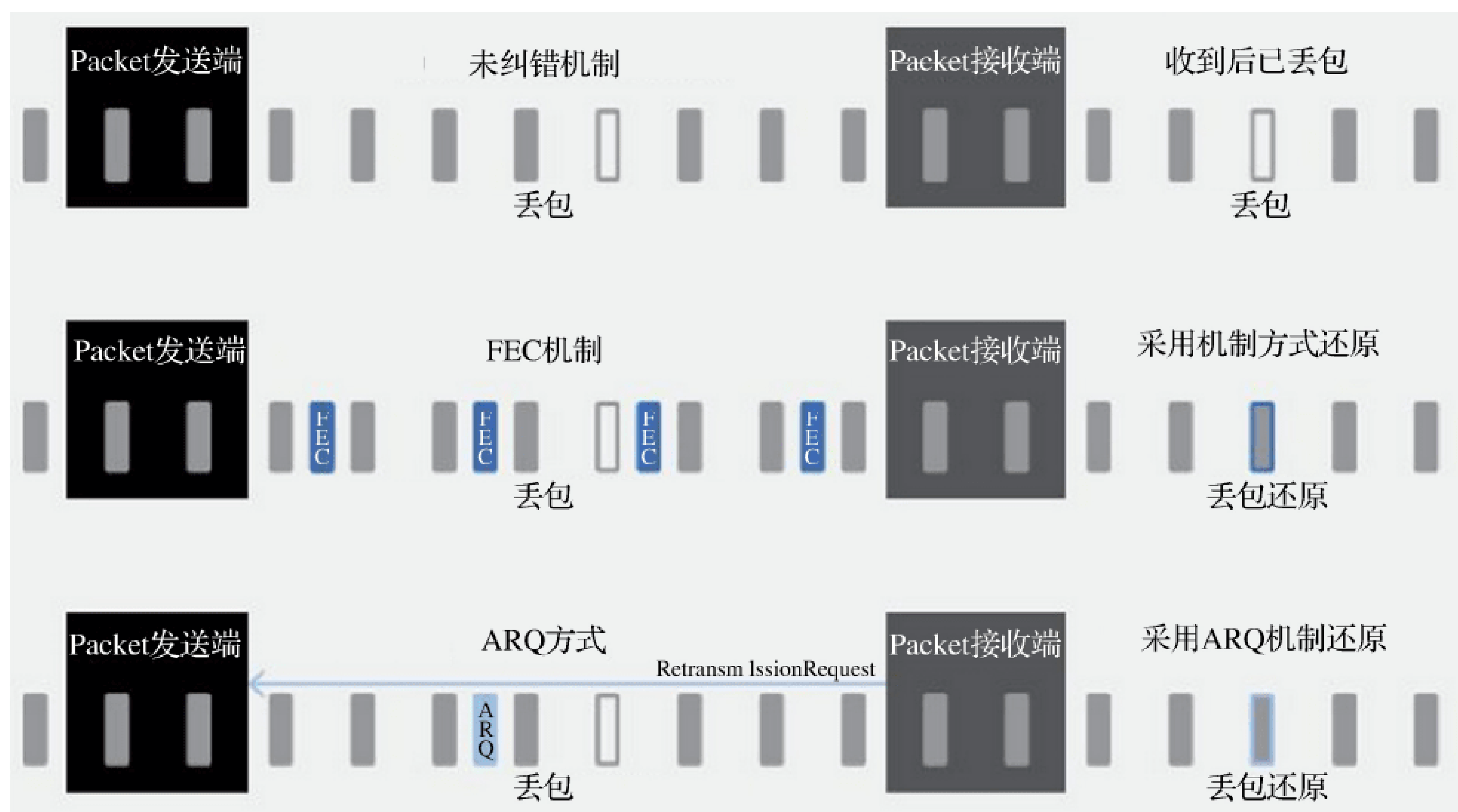


图 23－63　未纠错、FEC 与 ARQ 机制下的传输比较(图片来自 CSDN)

ARQ 是一种有的放矢的纠错机制。在正确的 ARQ 机制下每个数据包都会被赋予一个唯一的序列号,接收端根据这些序列号判断是否接收到了全部数据包。如果接收端在此过程中发现丢包则可以创建丢包序列号的列表并发回发送端,发送端通过这个列表重发丢失的包,在网络拥塞时这个过程可以重复多次。对比 FEC 机制,ARQ 机制提高了数据传输的有效性,在网络状态良好的情况下与未使用纠错策略的传输方式无异。

不过 ARQ 机制要求在发送端和接收端具备较大的缓冲区,一则是用于发送缓存数据包以便重发,二则也是为接收端对数据排序和纠错提供缓冲空间。因此 ARQ 机制是一种“以空间换时间”的机制。

23.3.1.2　SRTP/SRTCP

SRTP/SRTCP 是对 RTP/RTCP 协议的升级。SRTP(Secure Real-time Transport Protocol,安全实时传输协议)提供了传输数据加密、消息认证、完整性保护和重放保护等服务。与 RTP 一样,SRTP 也是基于 TCP/UDP 的应用层协议(SRTP over TCP/UDP),在实践中以基于 UDP 的方式居多。目前思科研发的 libSRTP 是 SRTP 协议的开源版本。

1. 撒盐加密

在介绍 SRTP 之前我们先介绍一下什么是撒盐(salt)加密。撒盐加密是一种保护对象机密性的加密方式,用接地气的语言来表述就是:当用户注册时,在密码上撒一把盐生成一种味道并记住这种味道;当用户再次登录时,在输入的密码上撒同一把盐也会生成一种味

道,此时闻一闻两种味道是否一致,一致则说明登录没有被仿冒。

在计算机系统中,用户注册时系统随机生成一个 salt 值,并将密码与 salt 连接起来产生散列值,最后将该散列值和 salt 值都存储于数据库中。用户登录时,系统会从本地的 Cookie 中寻找与用户名/密码匹配的 salt 值,并与用户密码进行连接与散列,以此来判断该散列值是否与注册时生成的散列值一致。

在撒盐技术中,“盐”(salt)是由系统生成的,即便两个用户使用了同一个密码,但由于系统为它们生成的 salt 值不同,因此他们的散列值也是不同的。就是靠这样一种原理实现了对于对象机密性的保护。

2. SRTP 报文格式

SRTP 分为认证区和加密区两大部分。认证区对应于 SRTP 报文的头域,这部分与 RTP 头无异。加密区则主要是指 RTP 负载区(payload)和尾部的扩展字段(padding),其中扩展字段包括一个可选字段和一个必选字段,如下所示:

➢ **SRTP MKI**:这是可选字段,表示主密钥标识符。

➢ **Anthentication Tag**:这是必选字段,代表了认证标签,在发送数据包时有用。

SRTP 包的报文格式如图 23-64 所示。

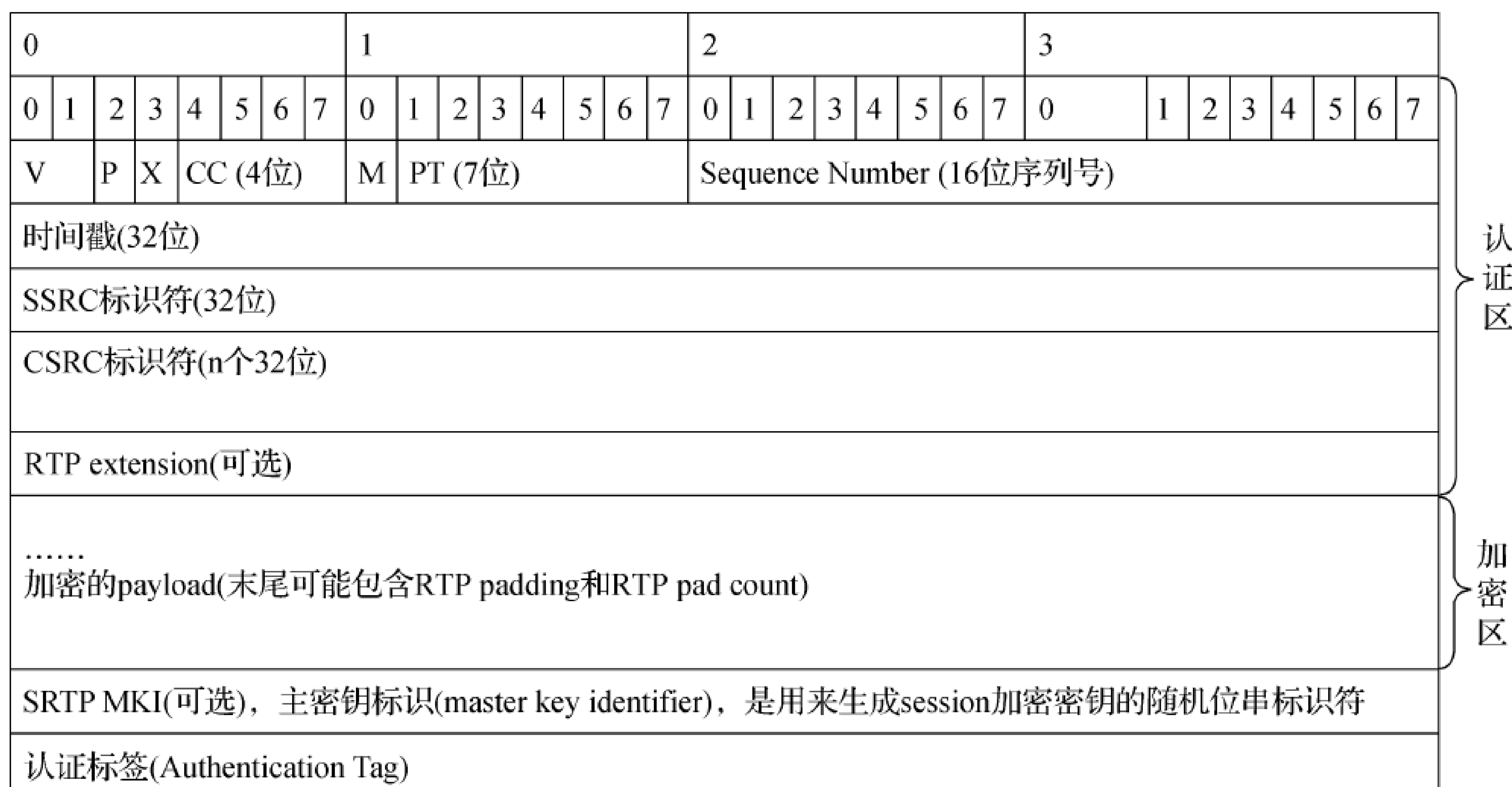


图 23-64 SRTP 报文格式

在传输时,SRTP 报文的认证区不加密,加密区可加密也可不加密。

3. SRTP 的加密

在 SRTP 中,发送端和接收端都要为每个 SRTP 流维护一份加密环境,这个加密环境就是加密的状态信息。每个加密环境都有自己的 ID,这个 ID 由同步源(SSRC)、目的地址和端口决定(Context ID = <SSRC DestAddress DestPort>),由此我们可以看出加密环境是与 SRTP 流绑定的。每个流中 ID 不匹配的 SRTP 包都会被丢弃。

加密环境保存了变换相关参数和变换无关参数。所谓变换相关参数是指与具体使用的加密/认证算法有关的参数,例如密码组大小、会话密钥、初始化向量数据等;变换无关参数自然就是与具体加密认证算法无关的参数了,例如循环计数器、序列号等,如图 23－65 所示。

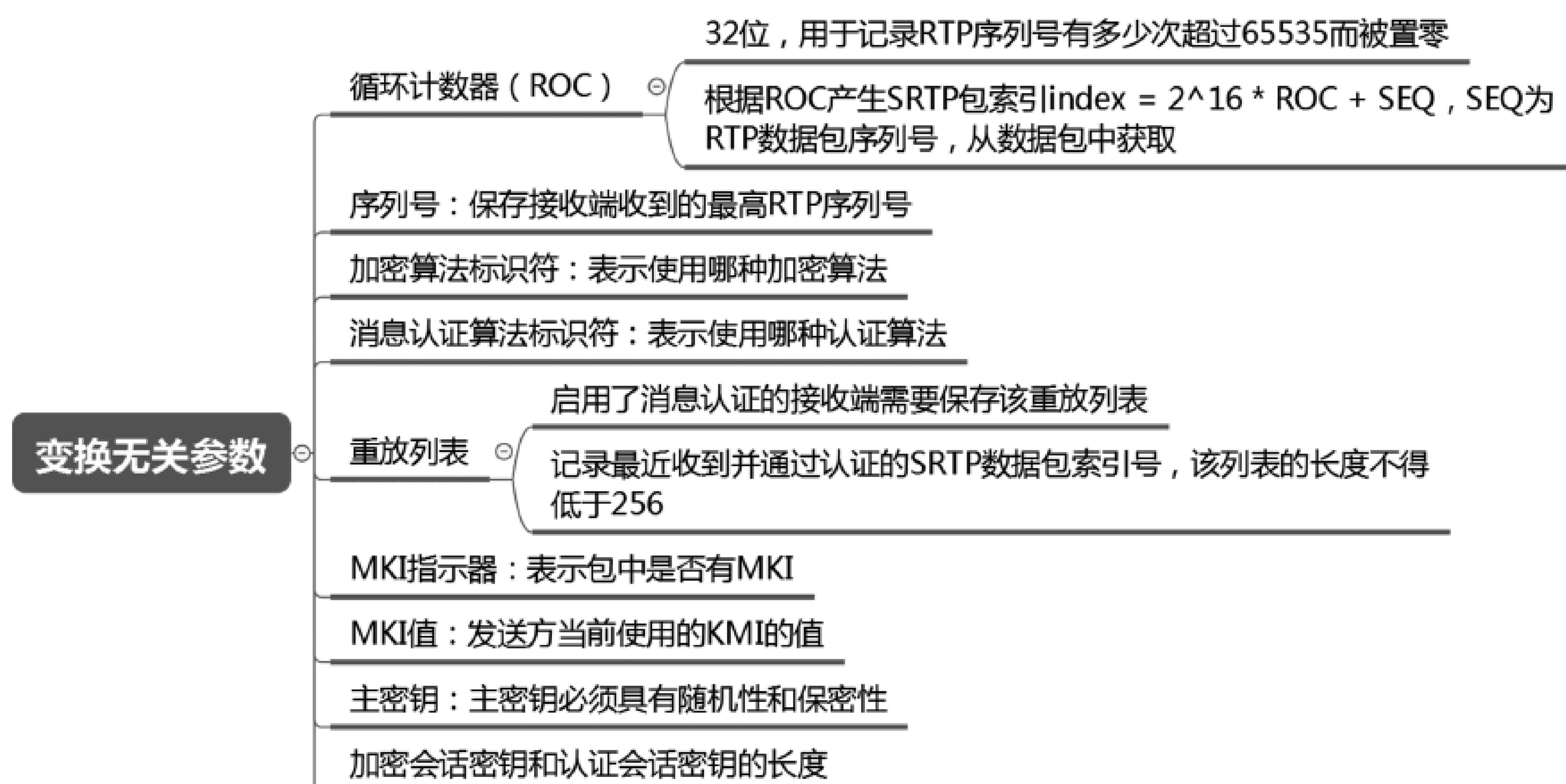


图 23－65　加密环境中的变换无关参数

SRTP 中的密钥分为主密钥(master key)和会话密钥(session key)两种。

主密钥用于生成相应的会话密钥。在一个加密环境下可以存在多个主密钥,至于处理 SRTP 包时到底使用哪个主密钥可以采用以下两种方式来做出决策:

- 由 SRTP 包中的 MKI 字段指定。这个字段原本就是用于确定加密方式的,只是这个字段为可选字段,启用后会增加包的长度,从而挤压有效包内容的长度。
- 可通过加密环境中的 <From,To> 为每一个主密钥指定所处理数据包的索引范围,超出这个范围则主密钥失效。这种方式多应用于单向和双向通信中,多向通信一般采用前一种方式。<From,To> 是由两个 48 位的时间戳组成的,表示主密钥的有效期(起止时间)。

会话密钥在加密传输时使用,用于 RTP 包的负载加密(加密变换)或消息认证。会话密钥由主密钥、主 salt、密钥推导率、会话密钥长度和 SRTP 包的索引号决定,会话密钥必须使用密钥生成器生成,如图 23－66。

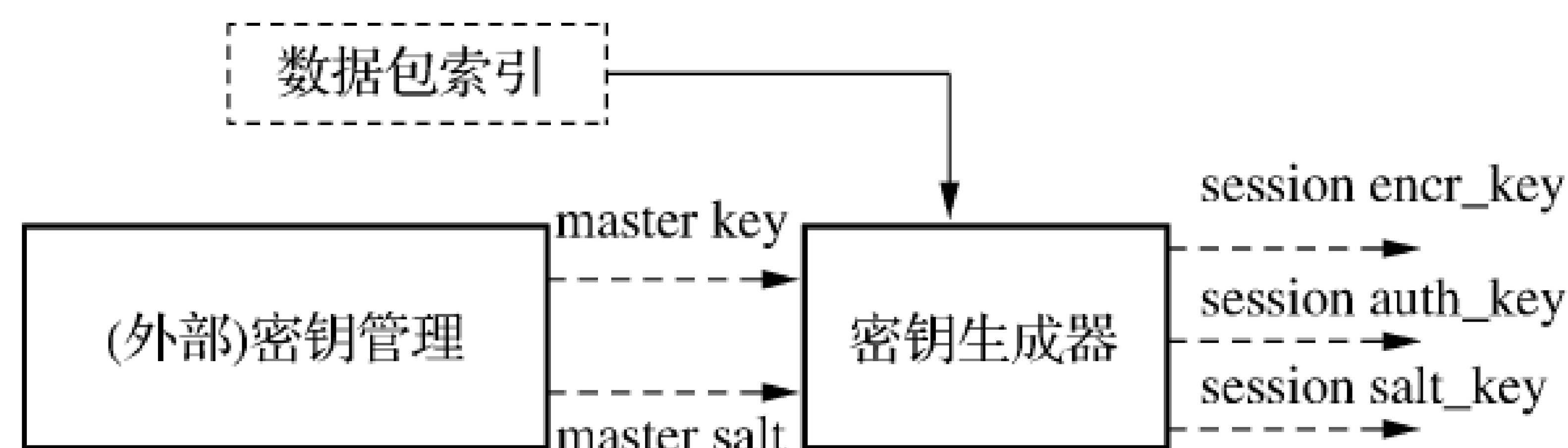


图 23－66　SRTP 中各种密钥的产生过程

SRTP 默认使用 AES 作为加密算法,使用 HMAC-SHA1 算法进行消息认证和完整性保护。

我们先来看看加密过程中的关键信息:

- **key_derivation_rate**:生成会话密钥的速率,必须是{1,2,4,…,2^24}中的一个,且必

须为 2 的幂数。

➢ **ROC(rollover counter)**:记录序列号的重置次数,用来计算 SRTP 包的索引,index = 2^16 * ROC + SEQ。

下面我们具体考察数据传输时对于消息的加解密过程。

(1) 发送端加密流程如下:

① 确定加密上下文。

② 确定要发送的 SRTP 数据包的索引。

③ 根据 SRTP 包的索引确定主密钥和主 salt。

④ 根据密钥管理协议中的参数集(主密钥、主 salt、密钥推导率等),通过密钥生成器生成会话密钥和会话 salt。

⑤ 使用加密上下文中指定的加密算法、会话密钥、会话 salt 对 RTP 的负载进行加密。

⑥ 若 MKI 标志位为 1,则加入 MKI 字段。

⑦ 对于消息加密,使用当前的 ROC、加密上下文中指定的加密算法和会话密钥计算认证标志,并填充 SRTP 包的认证标签字段。

⑧ 更新 ROC 和数据包索引。

(2) 接收端解密流程如下:

① 确定加密上下文。

② 根据公式计算收到的 SRTP 数据包的索引。

③ 确定主密钥和主 salt:通过 MKI 标志确定,或者通过 SRTP 包的索引确定。

④ 根据主密钥、主 salt、密钥推导率等参数确定会话密钥和会话 salt。

⑤ 进行重放检查:利用包索引和重放列表来检查重播,若为重播数据包则丢弃并记录到日志中。

⑥ 执行认证检查,如果不匹配则丢弃并记录到日志中。

⑦ 利用会话密钥解密 SRTP 包中的负载。

⑧ 根据包索引更新 ROC、最大序列号等参数,必要时更新重放列表。

⑨ 从数据包中删除 MKI 和认证标签。

除了加密与认证,SRTP 也具备防重放功能。SRTP 建议接收端维护收到的包索引号,并将其与每个新收到的消息作对比,接收端只应接收过去没有被播放过的新消息包,接收端会根据这个原理进行重放检查。

4. SRTCP 包

SRTCP 包相当于 SRTP 的 RTCP 控制包,默认使用与 SRTP 包相同的加密上下文。SRTCP 也会独立维护一个单独的重放列表以防止重放攻击。SRTCP 与 SRTP 使用相同的主密钥,但是会话密钥是不同的,而且 SRTCP 对主密钥也是独立计数的。根据这个计数值可以获取使用该主密钥处理过的 SRTCP 包的数量。

SRTCP 包与普通的 RTCP 包一样,都具有 5 种类型(如表 23-8 所示),其报文格式如

图 23－67 所示。

表 23－8　SRTCP 包的 5 种类型

类型	缩写表示	用途
200	SR(Sender Report)	发送端报告
201	RR(Receiver Report)	接收端报告
202	SDES(Source Description)	源描述
203	BYE	结束传输
204	APP	特定应用

由于 RTCP/SRTCP 包必须以复合包(Compound Packet)的形式发送(目的是提高发送效率),因此每次发送至少要包含两个单独的 RTCP/SRTCP 包。复合 SRTCP 的建议格式如下:



图 23－67　单 SRTCP 报文格式

- **加密前缀**:需要对复合包加密时则进行前缀加密。
- **SR 或 RR**:在复合包中的第一个 RTCP 包必须为状态通告包,以便进行报文头校验。
- **额外的 RR 包**:如果接收统计信息源数目超过 31,则需要在初始的报告数据包后添加额外的 RR 包。
- **SDES**:在复合包中必须包括一个包含 CNAME 项(代表一个客户端的 ID,无论一个客户端有几路流,其 CNAME 均一致)的 SDES(Source Description,源描述)包,其他的 SDES 根据应用的需要都是可选项。
- **BYE 或 APP**:其他 RTCP 数据包(包括未定义的)可以以任意顺序添加到复合包中。但是 BYE 必须放在复合包的最后,且须包含要终止的流的 SSRC/CSRC(特约信源)值。

5. DTLS_SRTP

DTLS_SRTP 是 DTLS 的一种扩展形式,是一种将 SRTP 的加解密和 DTLS 的密钥交换与会话管理技术相结合的机制。因此,从 DTLS 的角度看,DTLS_SRTP 为应用数据提供了新的数据格式(SRTP/SRTCP);而从 SRTP 的角度看,它提供了一种新的密钥协商方法。由于密钥等参数是 DTLS 握手协商得到的,因此该过程是保密的,比常用的诸如 SDP 的方式更不易被破解。

DTLS_SRTP 具有以下特征:

- 应用层数据的加解密仍然是由 SRTP 完成的。
- DTLS 的握手过程为 SRTP 加解密过程协商使用哪种密钥。

- 除了应用数据的加密格式为 SRTP 之外，其他记录层的报文（例如控制信息等）仍然为普通的 DTLS 格式。
- 当发送 SRTP 格式的应用层数据时会直接跳过 DTLS 加密层而直接透传到传输层进行发送。

接收端接收到 DTLS_SRTP 报文后会进行解复用，这是在传输层实现的，解复用时根据包头特征（第一个报文的首字节）进行区分：

- 0～1：表明可能是 STUN 报文；
- 20～63：表明可能是 DTLS 记录层报文；
- 128～191：表明可能是 RTP/SRTP 报文。

23.3.1.3　音视频封装容器

1. 视频的流态和文件态封装

音视频分为两种状态：流态和文件态，其本质就是处于传输中的状态和落盘（内存或硬盘）后的状态。这两种状态下对于音视频的封装要求是不一样的。以视频为例，流态下的封装机制要表现视频的序列性和时间性，也要表现出视频的负载特性和源特性，例如 RTP 封装视频时 RTP 头就要表现序列号、时间戳、负载类型、同步源（SSRC）等（如图 23－68 所示），RTCP 则需要反馈丢包等 QoS 信息。以上这些信息有助于接收端发现乱序和丢包，并同步播放速度、了解媒体编码种类以及发送者的标识等。

```
RTP     65 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=441, Time=576533195, Mark
RTP   1455 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=442, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=443, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=444, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=445, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=446, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=447, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=448, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=449, Time=576533195
RTP   1456 PT=DynamicRTP-Type-96, SSRC=0x627F51AA, Seq=450, Time=576533195
```

图 23－68　RTP 对于流态视频的封装

这里要注意的是，并不是说 RTP 就不能用于封装文件态的音视频数据。例如在存储实时流媒体数据时就可采用 RTP 来封装，只要能将包和包之间界定清楚就没什么不可以的。

在文件态下一般就不需要表现序列号、同步源和负载类型这些信息了，但时间戳还是要保留的。另外文件态的视频是供播放器播放的，因此一般在文件的开头存放关于解码的信息，例如采用什么编码格式、分辨率和帧率、视频本身是否有封装、是否有音频和字幕、总播放时长等。而时间戳信息是一定要保留的，至少在视频本身的封装中要体现 DTS（解码时间戳）和 PTS（呈现时间戳），否则播放器可能无法按照正常帧率来解码和渲染。

典型的文件态封装结构如 FLV、MP4 等，被称为“视频容器”，如图 23－69 和图 23－70 所示。

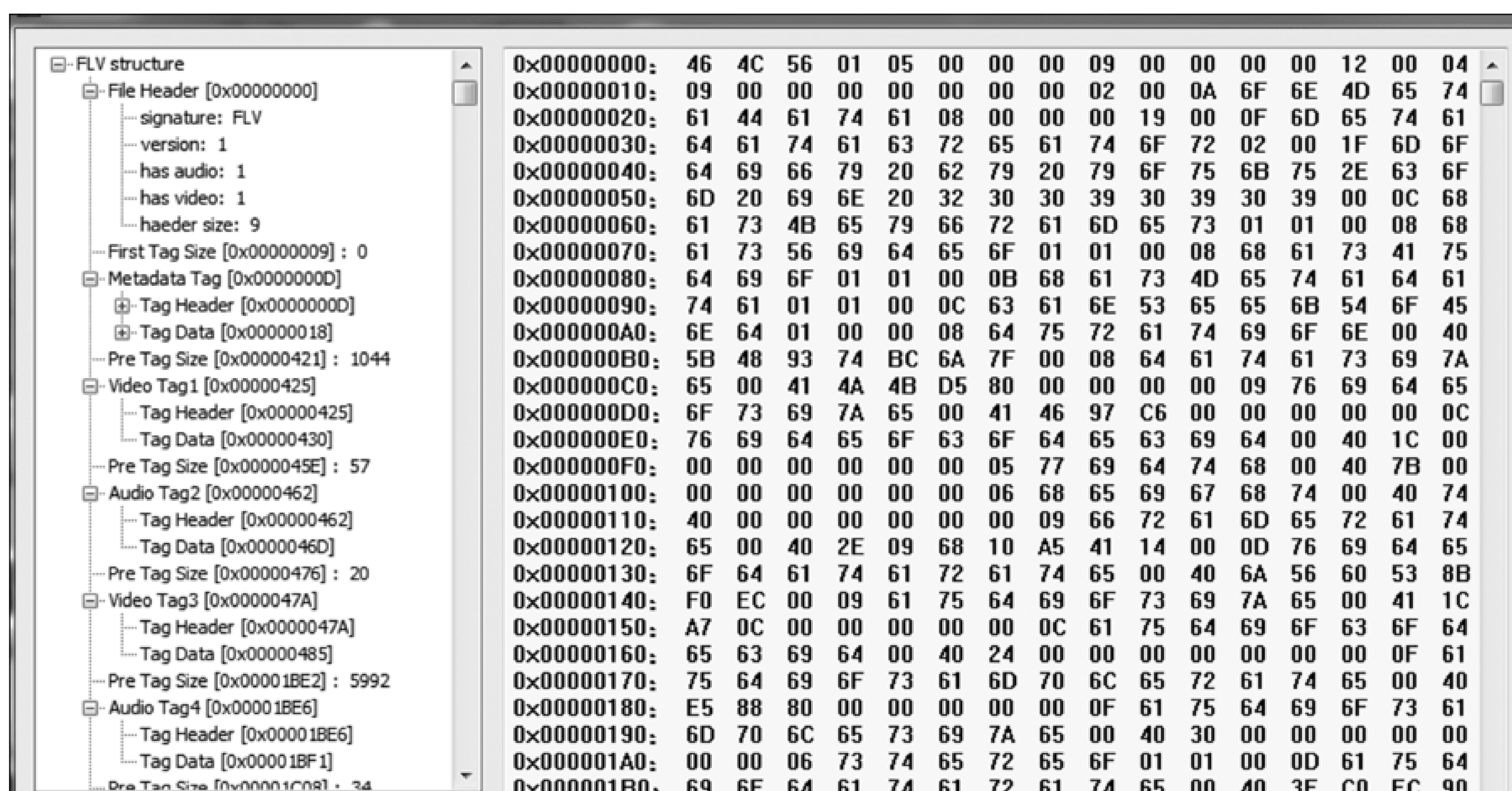


图 23－69　FLV 文件结构信息

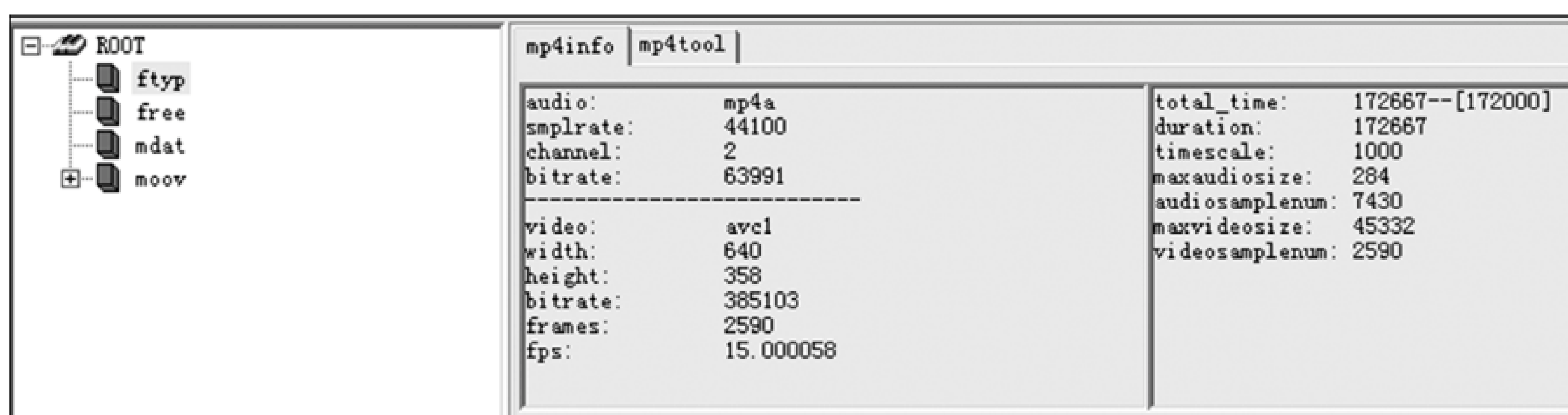


图 23－70　MP4 文件头信息

2. 视频的打包封装

视频还有一种封装方式，相对于上面两种容器更为底层（内层），这就是 PES/TS/PS 封装，有时也称为视频打包。打包封装居于流态或者文件态封装的内层。其中 **PES**（**Packet Elementary Stream**）称为打包的基本码流，这种码流也可看作对视频基本码流（Elementary Stream，ES）的封装；**TS**（**Transport Stream**）就是传输流，其每个包的长度都是相等的；**PS**（**Program Stream**）就是节目流，其每个包的长度不等。后两种是对视频数据的复合封装，也就是说 TS 和 PS 包中可以混合音频、字幕等信息，这种数据的流态也被称为“复合流”。

1）打包的基本码流（PES）

ES 首先被打包成长度不同的 PES 包，每个 PES 包的最大长度为 64 KB，其中开头为 6 B 的 PES 头，其结构如图 2－71 所示。

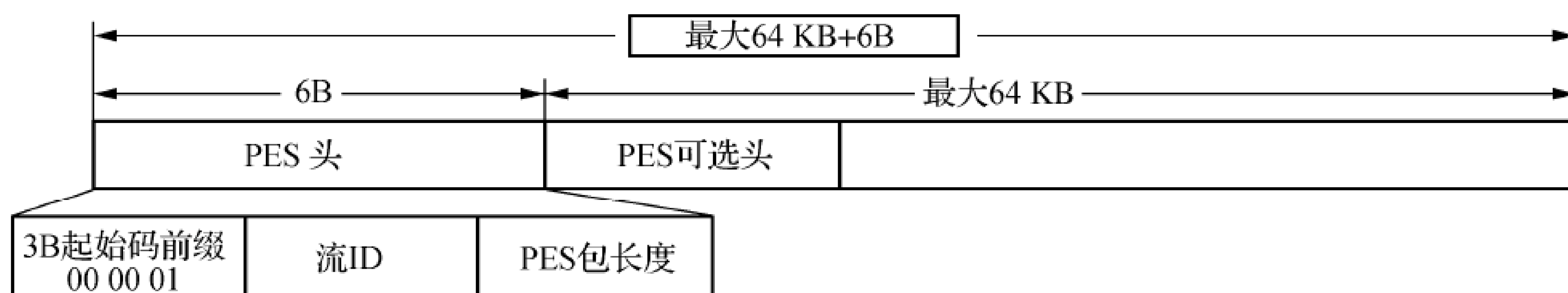


图 23－71　PES 包及包头结构





在 PES 头和 PES 媒体负载数据之间还有一个 Optional PES Header(PES 可选头),这里面存放了一些更重要的信息,这些信息对于解码有重要作用,例如 DTS 和 PTS 均存放在这个可选头结构中,如图 23－72 和图 23－73 所示。因此它虽然叫作"可选头",但还真是一点都不能少。

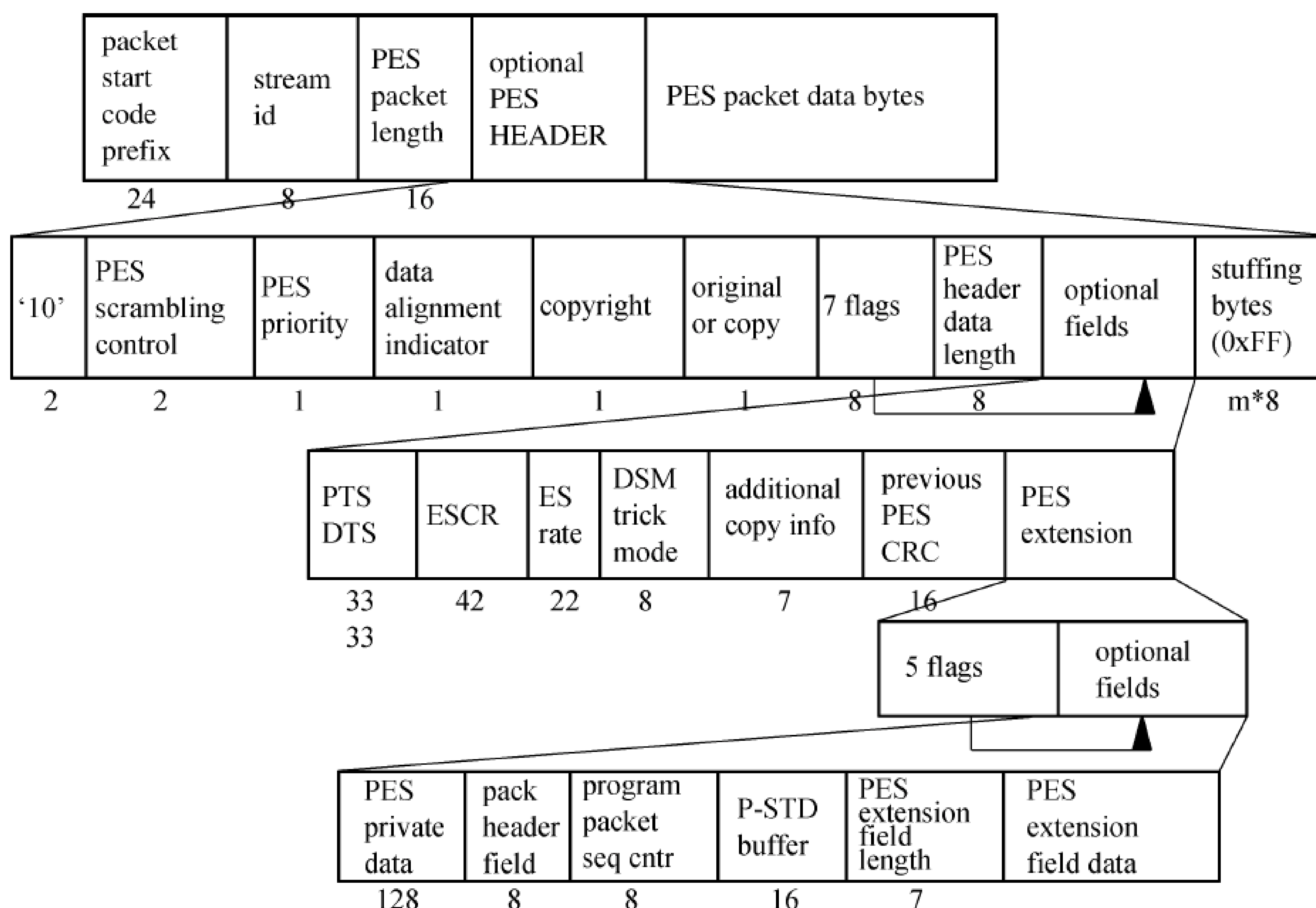


图 23－72　PES 可选头结构

```
0x000000F4 PES Packet (Video) { stream_id = 0xE0 }
        packet_length = 18
          PES_scrambling_control = 0
          PES_priority = 1
          data_alignment_indicator = 1
          copyright = 0
          original_or_copy = 0
          PTS_DTS_flags = 2
          ESCR_flag = 0
          ES_rate_flag = 0
          DSM_trick_mode_flag = 0
          additional_copy_info_flag = 0
          PES_CRC_flag = 0
          PES_extension_flag = 0
          PES_header_data_length = 7
            PTS = - 5: 8: 53: 117 (- 1 667 980 591)
```

图 23－73　实际视频文件中的 PES 包头和可选头结构

从 PES 头结构中我们可以看到起始码、流 ID(用以区分音频还是视频)和 PES 整个包体的长度等重要参数。PES 包的分割一般以帧为单位,如果某一帧小于等于 64 KB,则封装到一个 PES 包内;如果该帧大于 64 KB,则以这个最大值为步长分割并封装到若干 PES 包内。因此我们经常看到高清视频流的关键帧会被分成多个 PES 包。

2）传输流(TS)

TS 的全称是 MPEG2-TS,主要应用于实时传送的节目,比如实时广播的电视节目。这是

因为 TS 的特定信息包中携带频道信息，可以基于这些信息区分视频流隶属于哪个频道，其中 TS 包头中的数据包识别号就代表了频道信息，解复用时非常方便。

MPEG2 标准中规定 TS 传输包的长度为 188 字节，其中包头为 4 字节，负载为 184 字节，如图 23－74 所示。但通信媒介会为包添加错误校验字节，从而有了多于 188 字节的包长。



图 23－74　TS 包的包头结构

在 TS 中还包括了几种专有的数据包，这些包的信息描述了节目解复用相关的重要信息，被称为节目专有信息（PSI），这些信息包可以管理各种类型的 TS 数据包。DVB 标准对 PSI 进行了扩展，称为 SI。PSI 具有以下几种形式（SI 则比以下形式还要丰富）：

➢ **节目关联表（Program Association Table，PAT）**：PAT 给出了一路 TS 包含了多少套节目，并同时给出了它与 PMT 中 PID 的对应关系，因此在多路复用中最为重要。

➢ **节目映射表（Program Map Table，PMT）**：PMT 给出了每套节目中具体的组成情况与音视频 PID 的对应关系，在多路复用中也非常重要。

➢ **条件接收表（Conditional Access Table，CAT）**：用于将一个或多个专用的 ECM（授权控制信息）/EMM（授权管理信息）流分别与唯一的 PID 相关联。

➢ **网络信息表（Network Information Table，NIT）**：用于描述整个网络，例如包含了多少路 TS、这些流的频点和调制方式等信息。

SI 则还包括 SDT（业务描述表）、BAT（业务群关联表）、EIT（事件信息表）、RST（运行状态表）、TDT（时间和日期表）等信息。在广电业务中一般也采用 SI 作为专有信息载体，因为其更为丰富。

图 23－75 是一个 PAT 和 PMT 关系的例子，可以看出该 TS 中包含了 4 路节目（Program），每一路节目都作为一个 Program 包含在 PMT 中。图 23－76 则展示了一个 TS 封包的实例。

3）节目流（PS）

在 PS 方式封装的 H.264 视频流中，一般将 SPS、PPS、SEI、IDR 等 Nalu 封装为一个包，例如将 SPS、PPS、SEI 和 I 帧的开始部分等信息统一封装为一个 PES 包。PS 包会对 PES 包进行外层包装，由于 PS 包没有限制长度，因此大多数时候可以将一个大帧（例如 SPS、PPS、SEI ＋整个 I 帧）所分成的所有 PES 包封装成一个 PS 包以减少解封装的复杂度。同时，当包含音频数据时可以直接将音频封装成 PES 包附加在视频 PES 包的后面。

对于关键帧和非关键帧，PS 包封装的顺序是不同的：

➢ **关键帧的 PS 封包层次顺序为**：PS 头|PS 系统头|PS 系统映射（System Map）|PES 头|H.264 Raw Data。其中系统头在当且仅当 PS 包是每一个数据包时才存在，也被称为"系统标题"。

➢ **非关键帧的 PS 封包层次顺序为**：PS 头|PES 头|H.264 Raw Data。

在 PS 系统映射中（stream_type 字段）定义了具体的音视频流类型，其取值为 0x10（MPEG4 视频流）、0x1B（H.264 视频流）、0x80（SVAC 视频流）、0x90（G.711 音频流）、0x92（G.722.1 音频流）、0x93（G.723.1 音频流）、0x99（G.729 音频流）、0x9B（SVAC 音频流）。

PS 头和系统头的结构如图 23－77 和图 23－78 所示。

PES/TS/PS 这几种打包方式的外层既可以采用 RTP 或容器方式封装，也可以采用其他私有定制的方式封装，还可以不封装而直接传输，这取决于通信双方的具体业务要求，并没有硬性规定。



图 23－75　PAT 与 PMT 的关系

```
0x00000000 Transport Packet { PID = 0x0, Payload = Yes (184), Counter = 0, Start indicator }
0x000000BC Transport Packet { PID = 0x1001, Payload = Yes (184), Counter = 0, Start indicator }
0x00000178 Transport Packet { PID = 0x100, Payload = Yes (176), Counter = 14, Start indicator }
           Program Association Table, version 0                                          3
           Program Map Table, version 0                                                  4
0x00000184 PES Packet (Video) { stream_id = 0xE0 }                                       5
0x00000192 H264 AUD                                                                      6
0x00000198 H264 Sequence Parameter Set                                                   7
0x000001B4 H264 Picture Parameter Set                                                    8
0x000001BC I slice  # 0                                                                  9
0x00000234 Transport Packet { PID = 0x100, Payload = Yes (184), Counter = 15 }          10
0x000002F0 Transport Packet { PID = 0x100, Payload = Yes (184), Counter = 0 }           11
```

图 23－76　实际视频文件中的 TS 封包格式

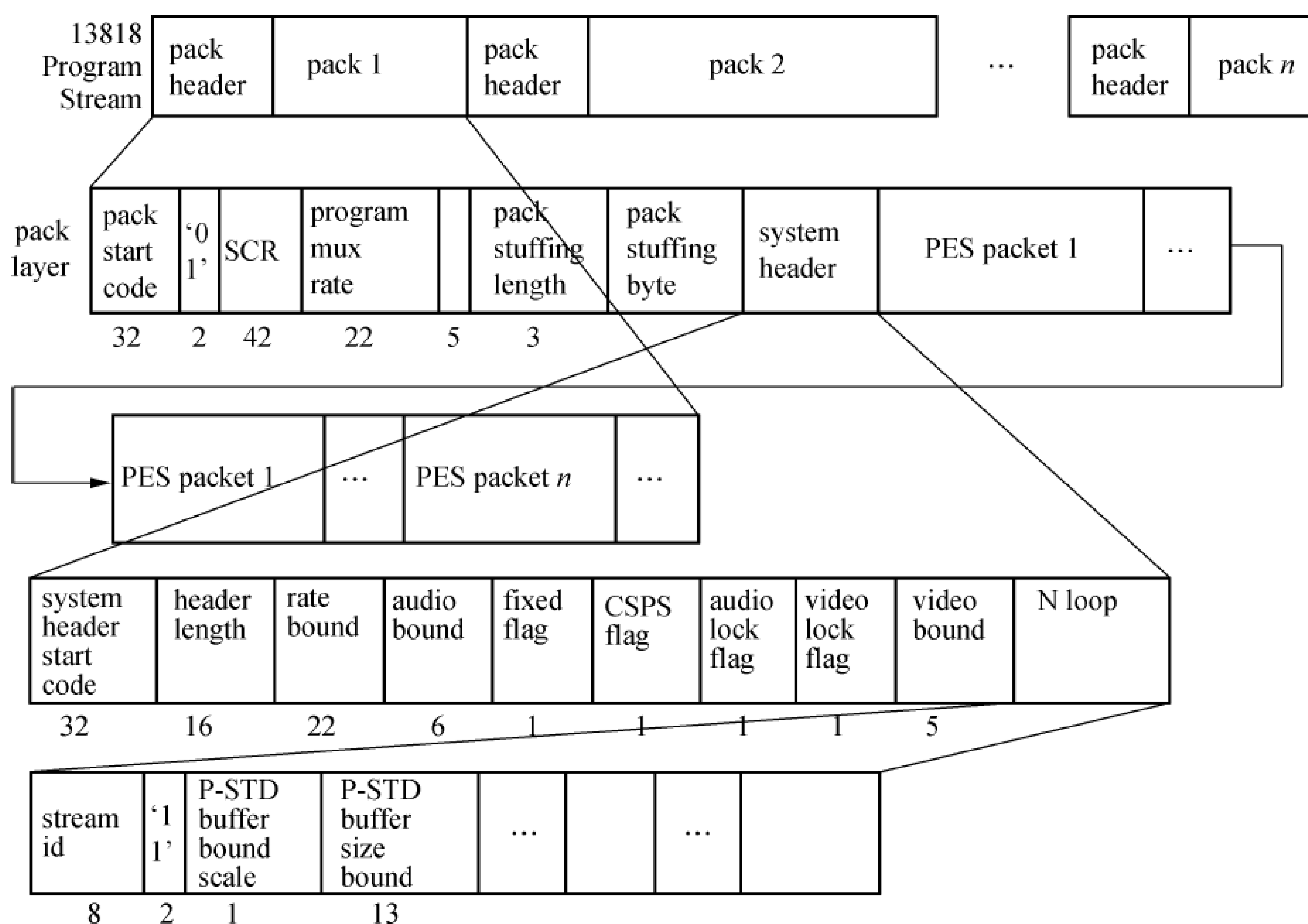


图 23－77　PS 头和系统头结构

```
StreamInfo
E:\各种抓包\example2.ps
FileSize:       1 335 572
StreamType:     Program
OverHead:       54 168 (10.12 %)
CountPackets:   500

Parsing progress ( complete )
[Stop]

Show Packets
[Show All] [Open All]
[Hide All] [Close All]
☑ Program stream
  ☑ PES (ID = 0xE0(224))
    ☑ ES (H264/AVC)

0x00000000 Program Pack { SCR =  2: 41: 12: 810 (870 552 932), MuxRate = 83 886 (4 194 300) }
0x0000000E System Header                                                       1
            header_length = 12
            rate_bound = 167772 (67108800)
            audio_bound = 1
            fixed_flag = 0
            CSPS_flag = 0
            system_audio_lock_flag = 0
            system_video_lock_flag = 0
            video_bound = 1
            packet_rate_restriction_flag = 0
              Stream 0xE0
              Stream 0xC0
0x00000044 PES Packet (Video) { stream_id = 0xE0 }                             2
0x00000057 H264 Sequence Parameter Set                                         3
0x0000007D H264 Picture Parameter Set                                          4
0x00000085 H264 SEI                                                            5
0x0000009D I slice  # 0                                                        6
0x00002057 PES Packet (Video) { stream_id = 0xE0 }                             7
0x0000406A PES Packet (Video) { stream_id = 0xE0 }                             8
0x0000607D PES Packet (Video) { stream_id = 0xE0 }                             9
0x00008090 PES Packet (Video) { stream_id = 0xE0 }                            10
0x0000A0A3 PES Packet (Video) { stream_id = 0xE0 }                            11
0x0000AC11 Program Pack { SCR =  2: 41: 12: 843 (870 555 902), MuxRate = 83 886 (4 194 300) }
0x0000AC1F PES Packet (Video) { stream_id = 0xE0 }                            13
0x0000AC32 H264 SEI                                                           14
0x0000AC42 P slice  # 1                                                       15
```

图 23－78　实际视频文件中的 PS 封包格式(注意 I 分片前面的 PS 头和系统头)





23.3.1.4 H.323 协议簇

H.323 是由 ITU-T 开发的基于 IP 网络的实时多媒体通信协议簇，也就是说，H.323 是个协议簇，而且是应用于多媒体的协议簇。这套协议簇由呼叫控制、流媒体编码、管理控制、网络安全等一系列相关的协议组成，体现了集中式控制和层次式控制的思想。H.323 协议簇多用于视频会议的接入与互联互通，是视频会议领域古老而知名的协议簇。由于发展历史较长，H.323 协议簇也比较庞大和成熟，特别是其可以与传统的电话网对接，扩大了其适用性。

1. H.323 协议栈框架

H.323 协议簇在本质上是个协议栈框架，这个框架包括了 H.245 控制、H.225.0 分组和同步、H.263 和 H.264 视频 CODEC、G.711/G.722/G.728/G.729 以及 G.723 音频 CODEC、T.120 系列多媒体通信协议等，如图 23－79 所示。

<table>
<tr><th colspan="2">数据</th><th colspan="2">信令</th><th>音频</th><th>视频</th></tr>
<tr><td>T.126</td><td>T.127</td><td colspan="2" rowspan="4">H.245
H.225.0
RAS</td><td rowspan="3">G.711
G.729
G.723.1
G.723.A</td><td rowspan="3">H.261
H.263
H.264</td></tr>
<tr><td colspan="2">T.324</td></tr>
<tr><td colspan="2">T.124 T.1245</td></tr>
<tr><td colspan="2">T.123</td><td colspan="2">RTP RTCP</td></tr>
<tr><td colspan="3">TCP</td><td colspan="3">UDP</td></tr>
<tr><td colspan="6">网络层</td></tr>
<tr><td colspan="6">链路层</td></tr>
<tr><td colspan="6">物理层</td></tr>
</table>

图 23－79 H.323 协议栈框架

H.245 是 H.323 终端的核心协议，承担了系统控制的功能，用于对 H.323 实体进行控制操作，包括呼叫控制（连接的建立与拆除）、通道切换与开放、描述逻辑信道内容等。因此，整个 H.245 协议由 H.245 控制通道、H.225.0 呼叫信令信道以及 RAS 信道组成。

H.245 是主叫网关和被叫网关之间的信息交互协议，分为四种类型：请求、响应、命令和指示。其中请求和响应用于协议实体，请求消息要求一个指定的行动以及一个立即的响应，而响应消息就是对请求的回复；命令消息要求一个指定的行动但不需要响应；指示消息只是提供信息，不要求指定行动和响应。

H.225.0 是 H.323 协议栈中解决数据流复用的方案，描述了在没有 QoS 保障的前提下对流媒体进行打包分组和同步传输的机制，同时也具有逻辑分帧、包编号、检错与纠错的能力。

RAS（Registration，Admission and Status，注册、管理与状态）是 H.225.0 协议簇定义的一种协议，是网关/终端与关守之间进行信息交互时使用的协议，包括注册、访问控制等内容，也就是说 H.323 实体通过 RAS 向关守进行注册。

Q.931 也是 H.225.0 定义的一种协议，其信令也是运行于网关与关守之间的，但它负责

的是呼叫过程中的信令处理，也就是说 H.323 实体通过 Q.931 完成了连接功能。

综上所述，RAS 完成实体向关守的注册，Q.931 完成实体间的连接功能（呼叫与被叫），H.245 完成连接实体间的参数协商（类似于 SDP），因此三者的执行顺序为 RAS→Q.931→H.245。

2. H.323 体系结构的组成

H.323 协议簇适用于在不提供额外 QoS 保障的情况下进行多媒体传输，实现了不同网络中终端之间的音视频交互。

H.323 定义了一系列的终端，包括网关（GW）、关守（GK）、多点控制器（MC）、多点处理器（MP）、多点处理单元（MCU）等，这里面最重要的是网关、关守和多点处理单元，这些终端设备可以呼叫其他终端，也可以被其他终端呼叫。由关守管理的网关、多点控制单元、多点控制器、多点处理器等所有终端组成的集合称为"域"。**一个域至少包含一个终端，且必须只有一个关守**。

下面我们来看一下构成域的这些节点和终端。

➢ **终端**：代表了能提供实时的双向语音通信能力（视频和其他数据的通信能力可选）的设备，也是一种用户终端，可以与 GW、MCU 等通信。终端是 H.323 系统中的必选组件，它必须包含以下能力，如图 23－80 所示：
   - 音频编解码能力是必备的，视频编解码能力可选；
   - 电子白板等数据应用能力；

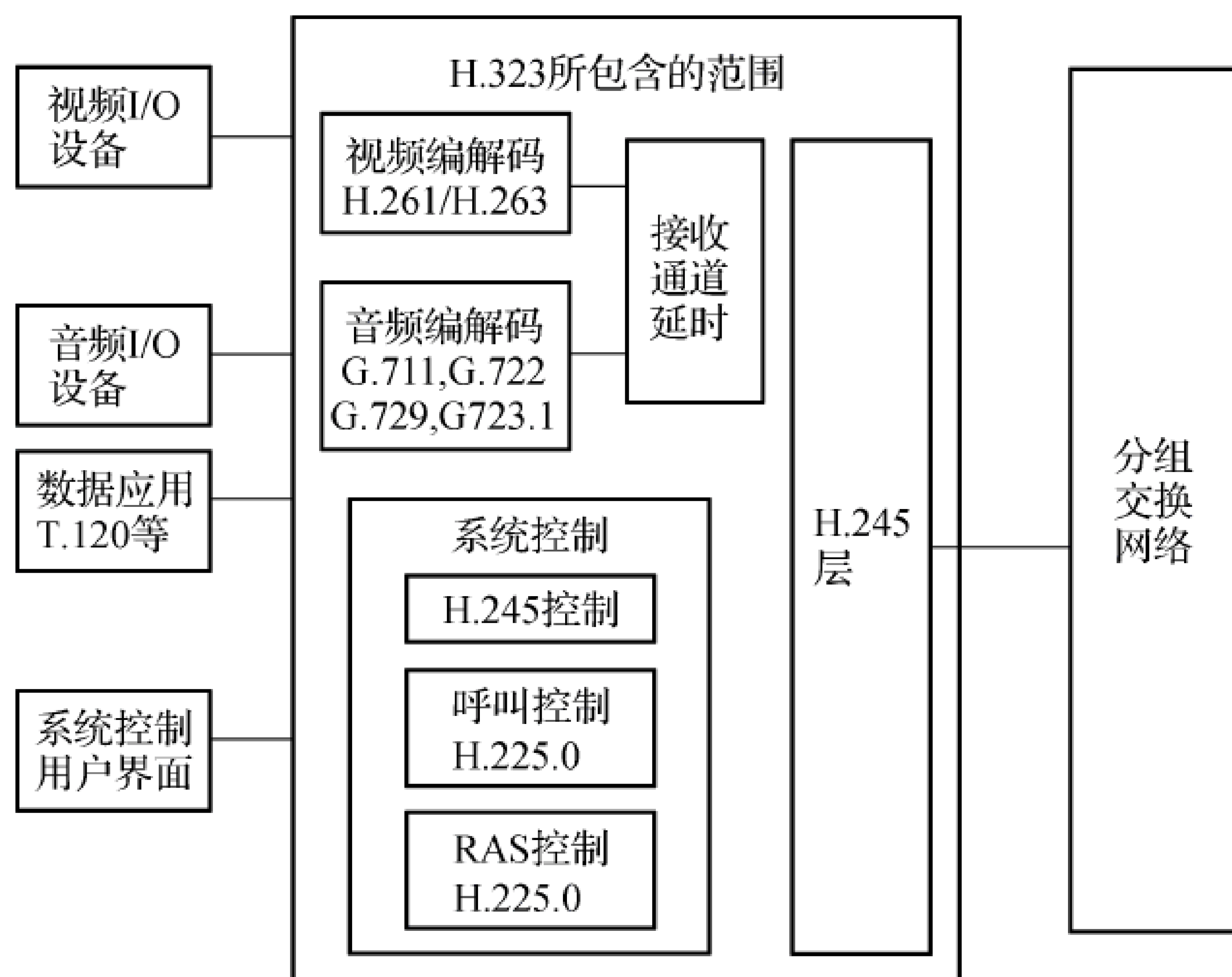


图 23－80　H.323 终端的能力组成

   - 提供端到端的信令（H.245 控制单元的能力），例如打开和关闭逻辑通道、发送指令以控制通信等；
   - 接收和发送音视频数据以及控制指令，同时也负责处理诸如逻辑分帧、增加序列号、错误检测等工作，这也是 H.225.0 协议定义的主要能力。

➢ **网关**:网关是 H.323 系统中的可选组件,提供了节点设备与其他兼容 ITU 标准的终端之间的协议转换功能(例如传输格式转换、通信协议转换等等)。同时,网关还可以执行音视频的转换工作,并且能够建立和拆除呼叫。

➢ **关守**:关守(gate keeper,也称网守)也是 H.323 系统中的可选组件,其功能是向节点设备提供呼叫接入控制等服务,当然这些服务功能也可以融入终端、网关和 MCU 中。关守定义了一些必选功能和可选功能,分别如下:

- 必选功能:地址翻译、带宽控制、接入控制、区域管理等;
- 可选功能:呼叫控制、呼叫鉴权、呼叫管理等。

在 H.323 协议体系中,由一个关守管理的所有终端、网关和 MCU 的集合称为 H.323 域。

➢ **多点控制单元**:MCU 支持三个以上节点设备的会议,由一个多点控制器 MC(必选)和几个多点处理器 MP(可选)组成,其中 MC 用于处理节点间的 H.245 控制信息,但并不处理媒体流;MP 则对音视频等媒体数据进行混合、切换与处理。MCU 在 H.323 域中的位置如图 23-81 所示。

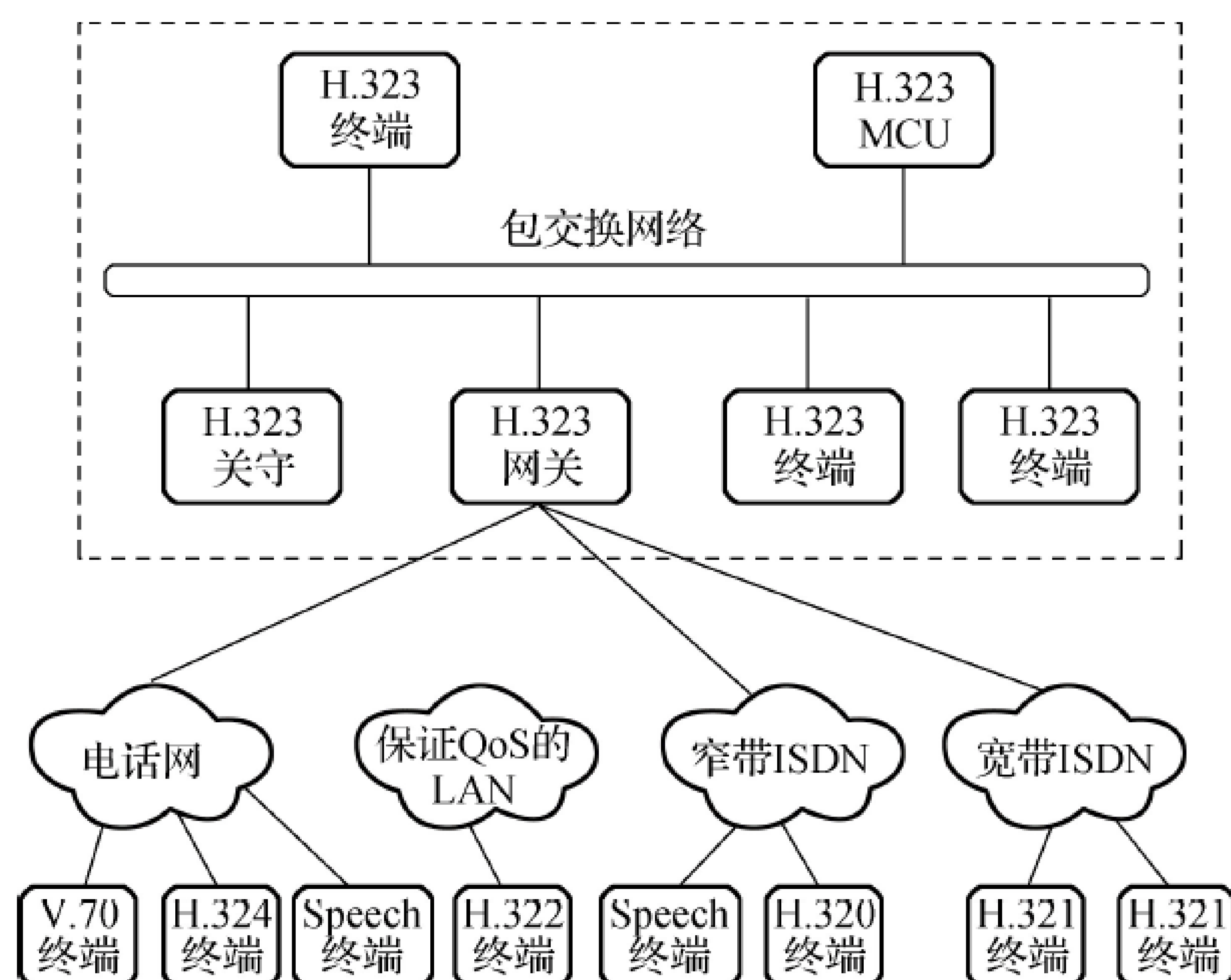


图 23-81　MCU 在 H.323 域中的位置

3. H.323 协议簇的执行流程

我们将以构成 H.323 协议簇的三个核心协议(RAS、H.245、Q.931)为例来描述会议系统的主要流程。

1) RAS 协议的执行流程

RAS 协议处理终端与关守之间或关守/网关与关守之间的注册与访问控制。前者由寻找关守消息、注册登记消息、注销消息、修改消息、接入认证授权和地址解析消息以及呼叫脱离消息构成;后者则由地址解析请求消息、状态消息、带宽改变消息、网关资源可利用性消息和 RAS 定时器修改消息构成。RAS 协议报文种类如表 23-9、23-10 和 23-11 所示。

表 23－9　关守/网关与关守间常用 RAS 协议相关消息

消息名称	消息全称	消息含义	消息分类
LRQ	Location Request	关守向上一级关守发出地址解析请求	地址解析请求消息
LCF	Location Confirm	上一级关守对 LRQ 消息的确认回答,并给出地址解析结果	
LRJ	Location Reject	上一级关守对 LRQ 消息的拒绝回答,并给出拒绝原因	
IRQ	Information Request	关守向网关发出的状态请求消息	状态消息
IRR	Information Request Response	网关根据 ACF 命令设定的间隔或 IRQ 向关守发送的状态回应消息	
IACK	Information Acknowledgement	对 IRR 消息的证实消息	
INAK	Information Negative Acknowledgement	对 IRR 消息的拒绝消息	
BRQ	Bandwidth Request	网关与关守之间的带宽改变的请求消息	带宽改变消息
BCF	Bandwidth Confirm	网关与关守之间的带宽改变的确认消息	
BRJ	Bandwidth Reject	网关与关守之间的带宽改变的拒绝消息	
RAI	Resource Availability Indication	网关向关守发送的资源可利用性报告	网关资源可利用性消息
RAC	Resource Availability Confirmation	关守对 RAI 消息的确认消息	
RIP	RAS Timers and Request in Progress	对 RAS 消息和后续的重试计数的响应	RAS 定时器修改消息

表 23－10　终端与关守间常用 RAS 协议相关消息

消息名称	消息全称	消息含义	消息分类
GRQ	Gatekeeper Request	受理终端初次使用,向网络广播寻找关守的请求,以找到自己所属的关守	寻找关守消息
GCF	Gatekeeper Confirm	关守向受理终端发送的对寻找关守请求(GRQ)的确认回答	
GRJ	Gatekeeper Reject	关守向受理终端发送的对寻找关守请求(GRQ)的拒绝回答	
RRQ	Registration Request	受理终端向关守发起的网关注册登记的请求	注册登记消息
RCF	Registration Confirm	关守向受理终端发送的对网关注册登记请求(RRQ)的确认回答	
RRJ	Registration Reject	关守向受理终端发送的对网关的注册登记请求(RRQ)的拒绝回答	

续表 23 – 10

消息名称	消息全称	消息含义	消息分类
URQ	Unregistration Request	受理终端向关守发送的关于网关请求注销注册登记的消息	注销消息
UCF	Unregistration Confirm	关守向受理终端发送的关于网关的 URQ 消息的确认回答;或计费认证中心向受理终端发送的关于用户的 URQ 消息的确认回答	
URJ	Unregistration Reject	关守向受理终端发送的关于网关的 URQ 消息的拒绝回答;或计费认证中心向受理终端发送的关于用户的 URQ 消息的拒绝回答	
MRQ	Modification Request	受理终端向计费认证中心发送的修改用户数据的请求	修改消息
MCF	Modification Confirm	计费认证中心向受理终端发送的对修改用户数据请求的确认消息	
MRJ	Modification Reject	计费认证中心向受理终端发送的对修改用户数据请求的拒绝消息	
ARQ	Admission Request	网关向关守发送的用户接入认证、地址解析请求消息	接入认证授权和地址解析消息
ACF	Admission Confirm	关守对 ARQ 消息的确认回答并给出地址解析结果,对于卡号用户,还需要给出用户余额和最长通话时长	
ARJ	Admission Reject	关守对 ARQ 消息的拒绝回答并给出拒绝原因	
DRQ	Disengage Request	网关与关守之间的呼叫脱离请求消息。当该消息由网关发起时,应同时传递计费信息。计费信息放在非标准数据(Non Standard Data)字段中	呼叫脱离消息
DCF	Disengage Confirm	关守对 DRQ 消息的确认回答	
DRJ	Disengage Reject	关守对 DRQ 消息的拒绝回答并给出拒绝原因	

表 23 – 11　顶级关守之间常用 RAS 协议相关消息

消息名称	消息全称	消息含义
业务请求	Service Request	顶级关守间的业务请求消息
业务确认	Service Confirmation	收到业务请求的顶级关守对 Service Request 消息的确认回答,并建立业务关联关系
业务拒绝	Service Rejection	顶级关守对 Service Request 消息的拒绝回答,并给出拒绝原因
描述器 ID 请求	Descriptor ID Request	顶级关守向别的顶级关守请求描述器 ID

续表 23－11

消息名称	消息全称	消息含义
描述器 ID 确认	Descriptor ID Confirmation	顶级关守对 Descriptor ID Request 消息的确认回答,并给出该顶级关守的描述器 ID 列表
描述器 ID 拒绝	Descriptor ID Rejection	顶级关守对 Descriptor ID Request 消息的拒绝回答,并给出拒绝原因
描述器请求	Descriptor Request	顶级关守向另一个顶级关守请求特定描述器的内容
描述器确认	Descriptor Confirmation	顶级关守对 Descriptor Request 消息的确认回答,并给出描述器的具体内容
描述器拒绝	Descriptor Rejection	顶级关守对 Descriptor Request 消息的拒绝回答,并给出拒绝原因
地址解析请求	Access Request	顶级关守间的地址解析请求消息
地址解析确认	Access Confirmation	顶级关守对地址解析请求消息的确认回答
地址解析拒绝	Access Rejection	顶级关守对地址解析请求消息的拒绝回答

RAS 协议的执行流程如图 23－82 所示,依次为关守发现、节点登记与注销以及呼叫接入与退出。由于比较简单,这里不再赘述。

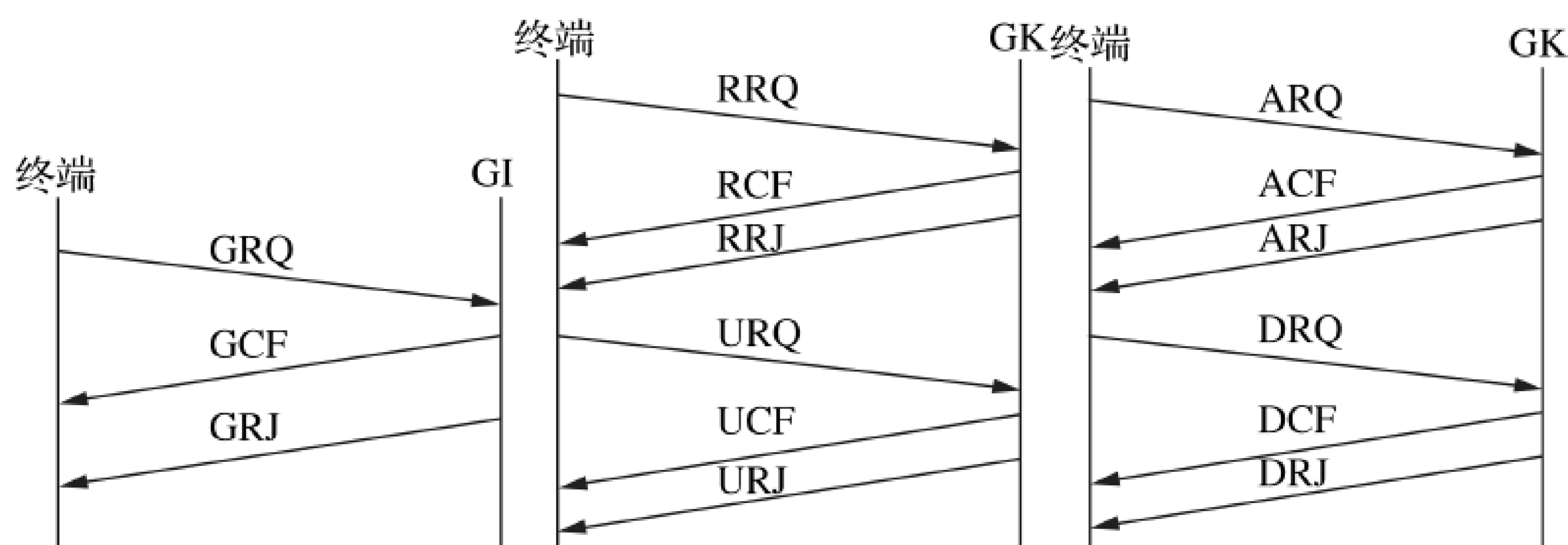


图 23－82　RAS 协议的执行流程

2）Q.931 协议的执行流程

Q.931 协议实现主叫方与被叫方的连接控制功能,是处理网关/顶级关守与关守之间信息交互所使用的协议,主要负责呼叫过程中的信令处理,其报文的种类如表 23－12 所示。

表 23－12　Q.931 消息类型

消息名称	消息功能	消息含义
Setup	呼叫建立	主叫发给被叫的消息,表示希望建立通话
Call Proceeding	呼叫进程	被叫发给主叫的消息,表示呼叫正在处理
Alerting	提醒	被叫发给主叫的消息,表示被叫用户已振铃
Progress	进展	用户或网络发送的消息,说明一个呼叫的进展情况

续表 23－12

消息名称	消息功能	消息含义
Connect	连接	被叫发给主叫的消息，表示被叫用户已摘机
Notify	通知	用户或网络发送的消息，用于对状态询问（Status Inquiry）消息进行响应或在呼叫期间对特定错误情况进行报告
Status	状态	顶级关守向另一个顶级关网请求特定描述器的内容
Status Inquiry	状态询问	用户或网络发送的消息，用于从一个同等的三层实体请求状态信息
User Information	用户信息	用户或网络发送的附加消息，用于提供呼叫建立或各种与呼叫相关的信息
Release Complete	释放完成	由先挂机的一方发给另外一方，表示释放过程已完成

Q.931 协议的执行分为两部分：呼叫建立和呼叫断开，我们先来看呼叫建立的流程。

呼叫建立最关键的就是找路由，可以采用两种方式查找路由，即直接路由方式和 GK 路由方式。

（1）直接路由方式的呼叫建立流程（如图 23－83 所示）

① 主叫终端发起呼叫，通过 RAS 协议的 ARQ 消息接入，在收到 GK 的 ACF 消息后解析出被叫终端地址，并直接与之建立 TCP 连接（被叫终端的监听端口一般为 1720）。

② 主叫终端通过 Q.931 协议发送 Setup 消息给被叫终端，被叫终端会回复 Call Proceeding、Alerting 和 Connect 等消息，建立 TCP 之上的会话连接。

③ 主叫终端收到 Connect 消息后，进入 H.245 协商阶段。

主叫终端和被叫终端都可以发出 Release 消息以结束本次呼叫会话。

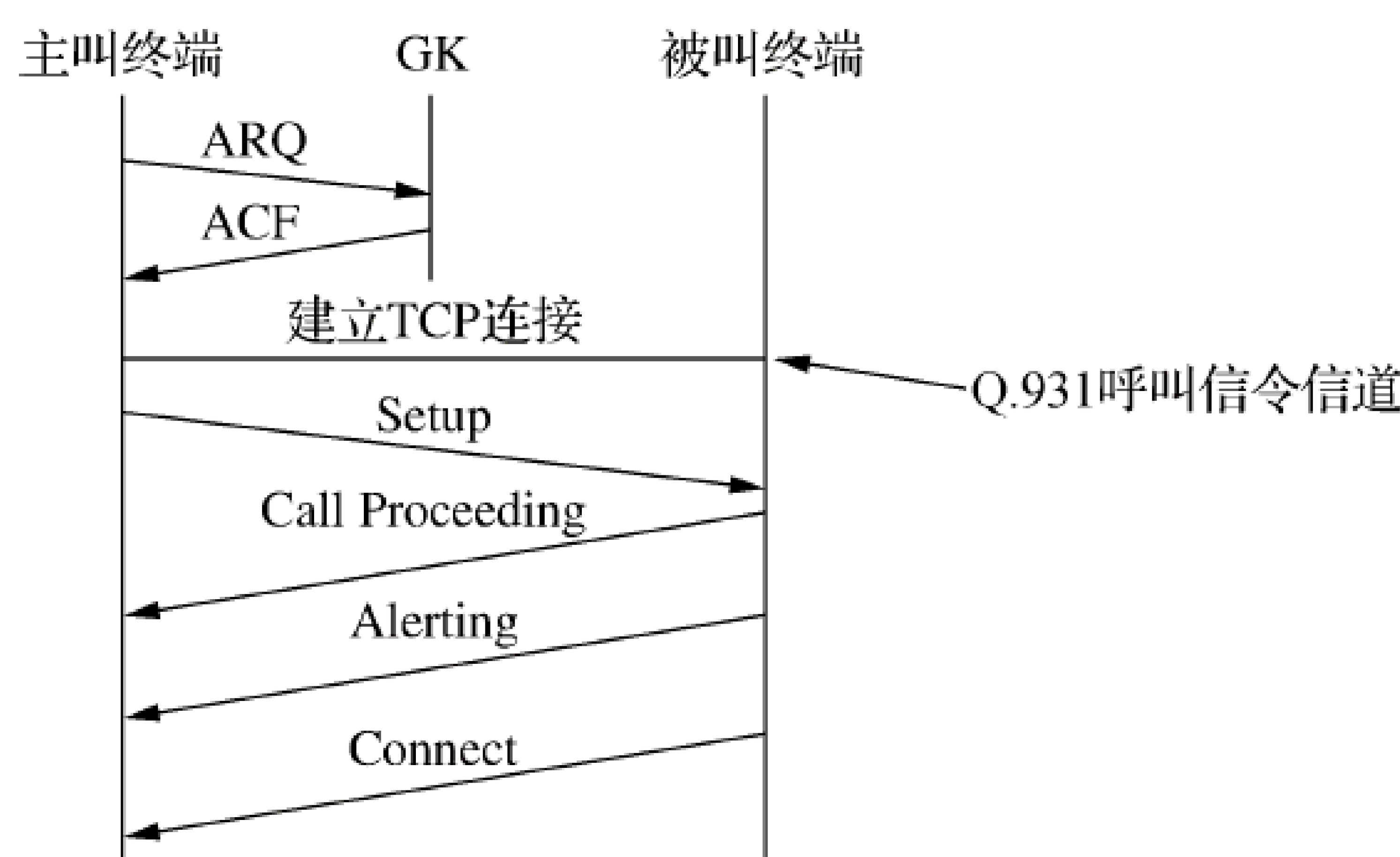


图 23－83　直接路由方式的呼叫建立流程

（2）GK 路由方式的呼叫建立流程（如图 23－84 所示）

① 主叫终端发起呼叫，通过 RAS 协议的 ARQ 消息接入，在收到 GK 的 ACF 消息后解析出被叫终端地址，这个地址是需要 GK 路由的，因此先与 GK 建立 TCP 连接（默认也是 1720 端口）。

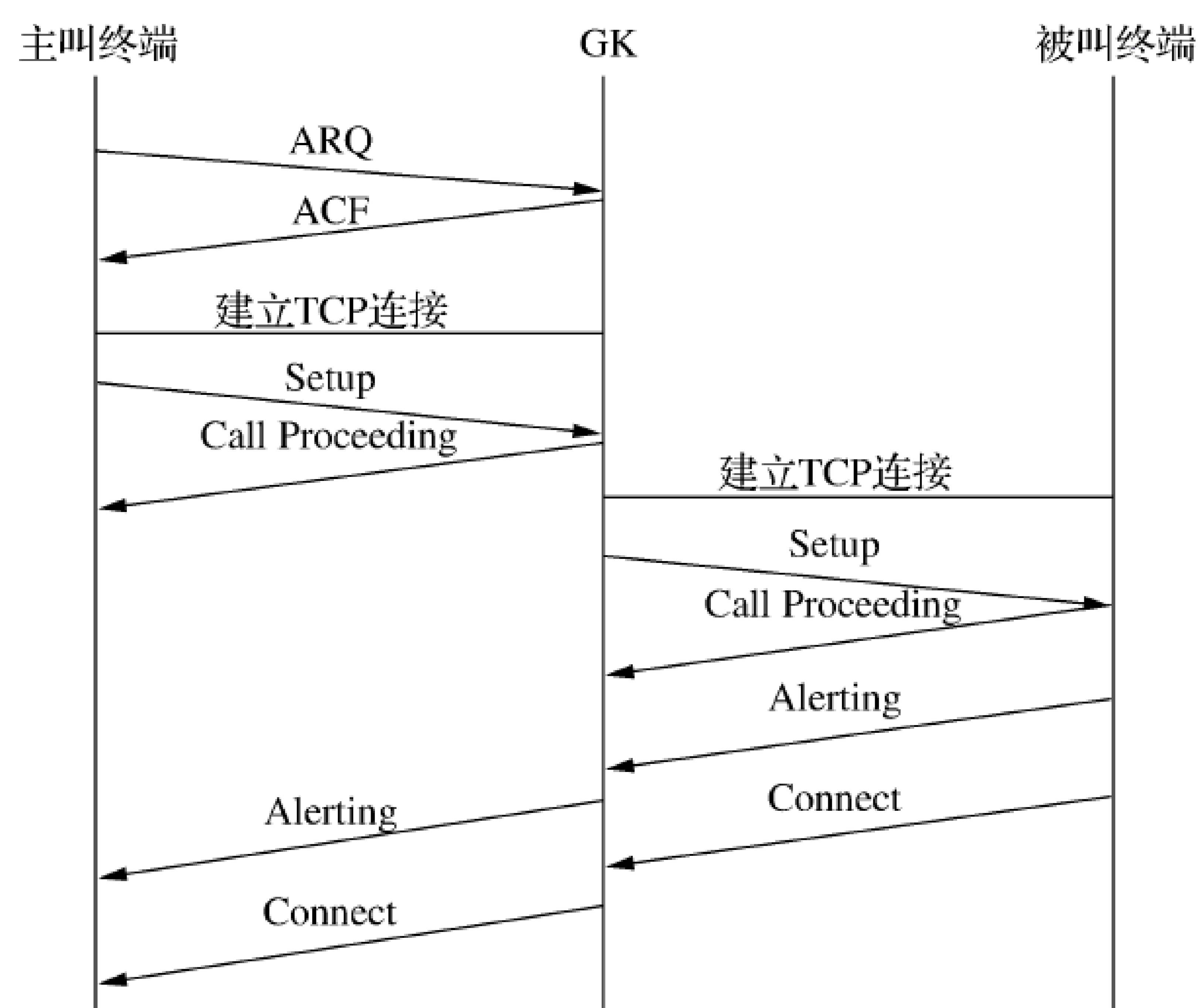


图 23－84　GK 路由方式的呼叫建立流程

② 主叫终端通过 Q.931 协议发送 Setup 消息给 GK，GK 会回复 Call Proceeding 消息，以代替被叫终端与主叫终端建立 TCP 之上的会话连接。

③ GK 与被叫终端建立 TCP 连接并发送 Setup 消息给被叫终端，被叫终端一般会回复 Call Proceeding、Alerting 和 Connect 等消息以建立会话连接。

④ GK 将上述收到的 Alerting 和 Connect 消息返回给主叫终端。

⑤ 主叫终端收到 Connect 消息后，进入 H.245 协商阶段。

主叫终端和被叫终端都可以发出 Release 消息以结束本次呼叫会话。呼叫断开的时候，无论主叫和被叫谁先挂机，都会发送 Release Complete 消息给对端并断开 TCP 连接，详见图 23－85。

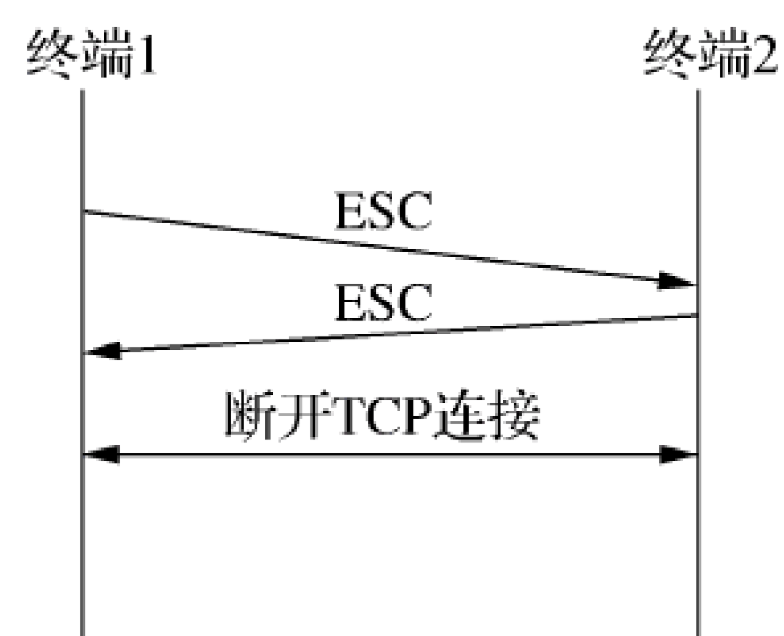


图 23－85　呼叫断开流程

3）H.245 协议的执行流程

H.245 协议负责主叫与被叫之间的信息交换，分为请求、响应、命令、指示 4 类消息。H.245 协议的控制功能主要由以下三个因素来考量：

- **主从问题**：决定谁是主、谁是从，可以解决会议中的某些冲突情况，例如两个节点都是 MC 或者两个节点都试图创建双向信道，这时需要决定谁是主、谁是从。
- **能力交换**：主要是通信双方的编解码能力的协商。
- **逻辑通道开关**：即通过 RTP/RTCP 为通话建立逻辑信道。

H.245 协议报文的种类如表 23－13 所示。

表 23－13　主叫与被叫间的 H.245 消息

消息名称	消息全称	消息含义	消息分类
TCS	Terminal Capability Set	能力交换请求，告诉对方本端支持的接收能力	终端能力设定
TCSA	Terminal Capability Set Acknowledge	能力交换请求响应	
TCSR	Terminal Capability Set Reject	能力交换请求拒绝	
MSD	Master Slave Determination	主从确定请求	主从决定
MSDA	Master Slave Determination Acknowledge	主从确定请求响应	
MSDR	Master Slave Determination Reject	主从确定请求拒绝	
OLC	Open Logical Channel	打开逻辑通道请求	打开逻辑通道
OLCA	Open Logical Channel Acknowledge	打开逻辑通道请求响应	
OLCR	Open Logical Channel Reject	打开逻辑通道请求拒绝	
ESC	End Session Command	结束会话命令，即关闭 H. 245 通道	结束会话
CLC	Close Logical Channel	关闭逻辑通道命令	关闭逻辑通道
CLCA	Close Logical Channel Ack	关闭逻辑通道响应	

H. 245 协议的执行流程也非常简单，从图 23－86 至图 23－88 中可以看出基本上是一来一回的同步式请求握手。

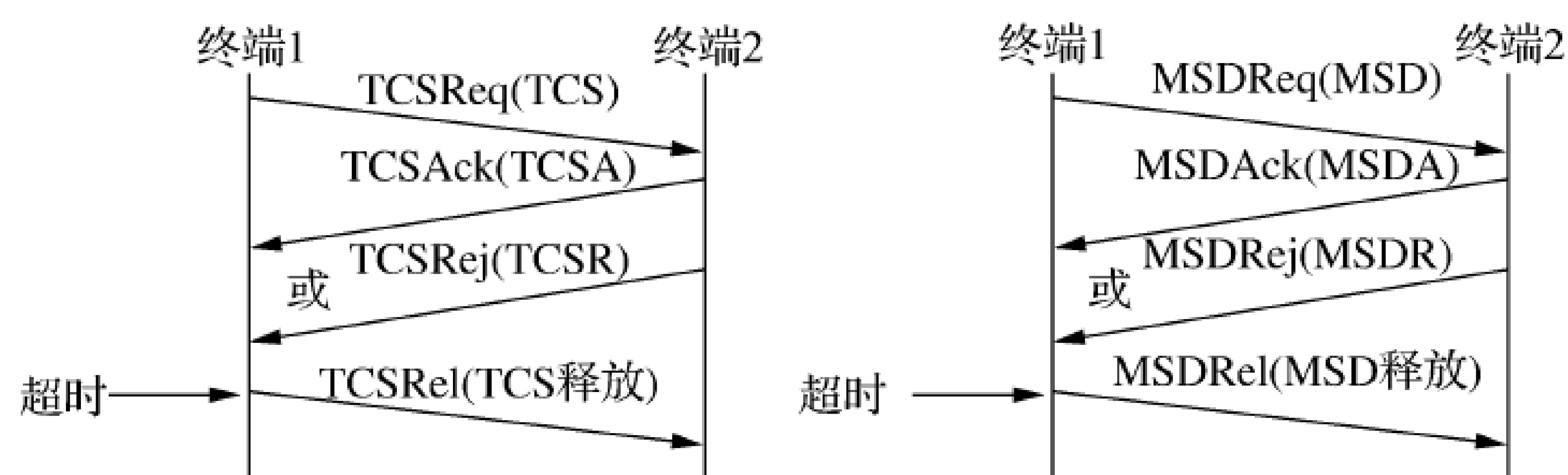


图 23－86　H.245 协议中能力交换与主从确定的流程

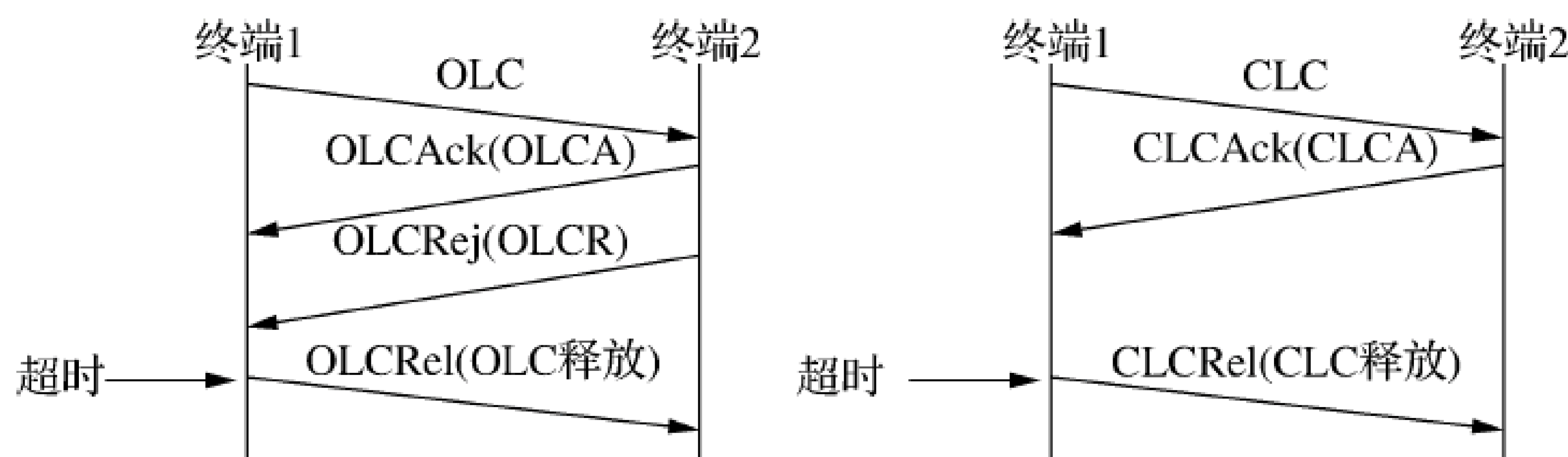


图 23－87　H.245 协议中打开逻辑通道与关闭逻辑通道的流程

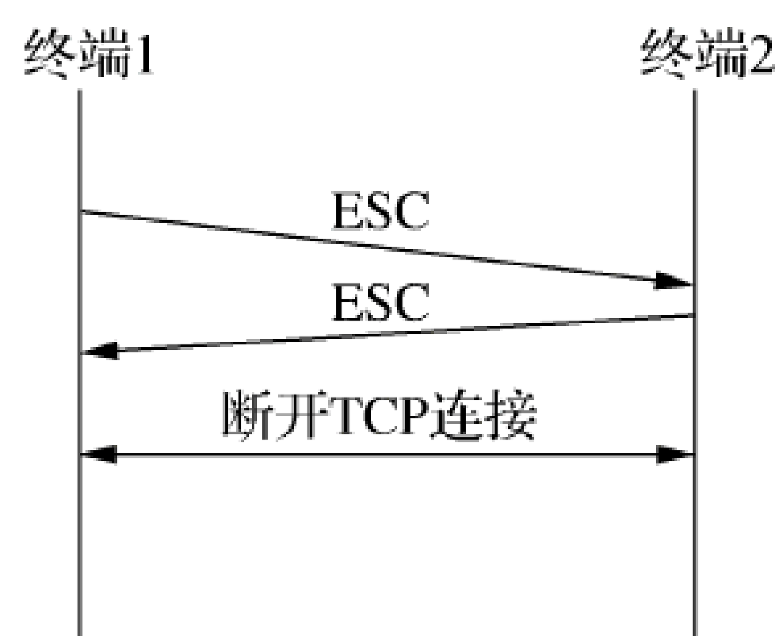


图 23－88　H.245 协议中结束会话的流程

综上所述,我们可以梳理出典型的完整呼叫和断开流程,如图 23－89 所示。

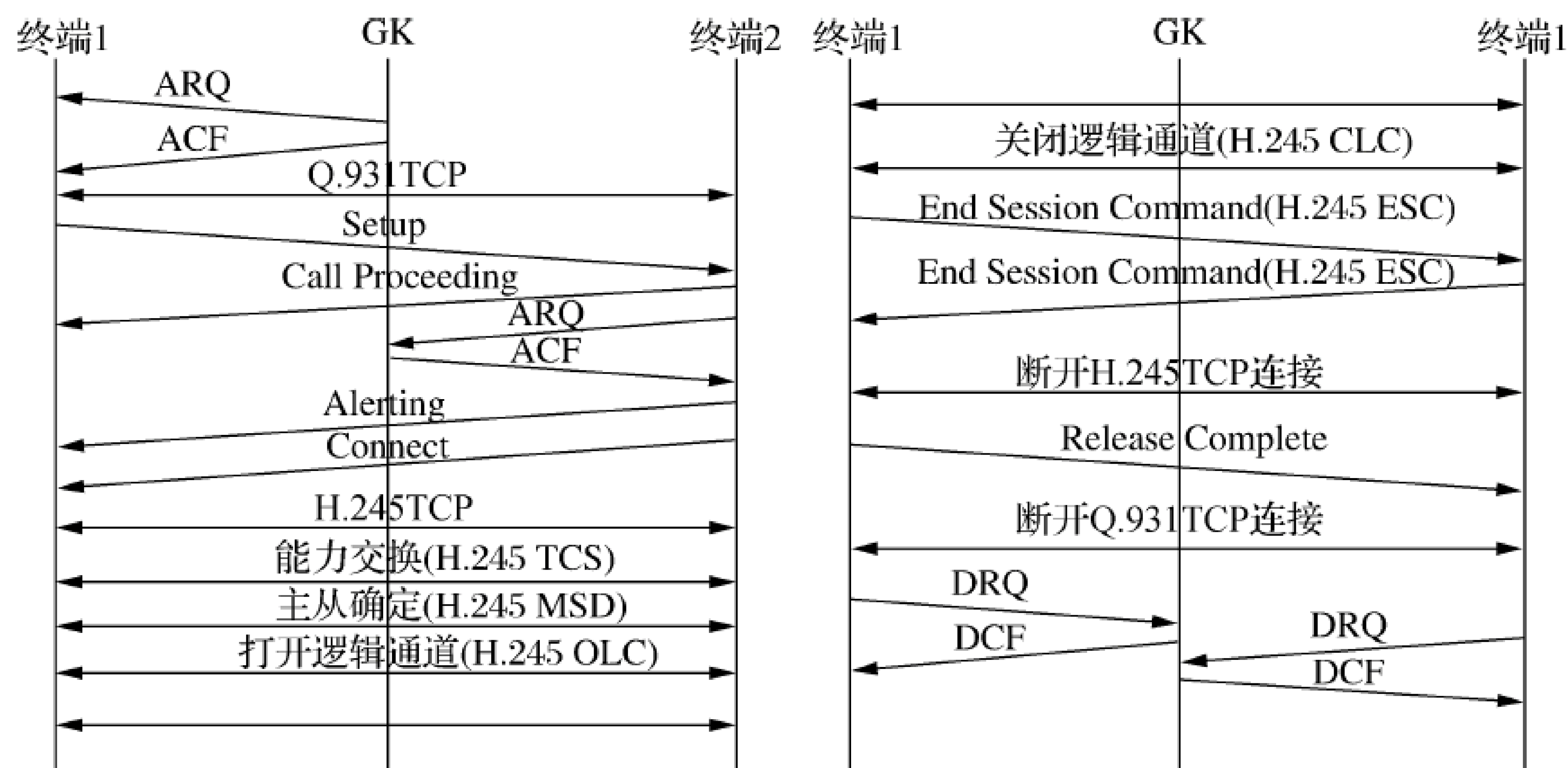


图 23－89　典型的呼叫和断开流程

23.3.1.5　GB/T 28181 和 GB 35114

近年来随着视频图像互联互通要求的日益迫切,公安部密集出台了一系列关于互联互通技术的标准规范,包括技术标准、评定标准、管理标准等内容。其中 GB/T 28181 规范在整个标准体系中占据了核心地位,而 2018 年发布的 GB 35114 规范又是 GB/T 28181 在数据安全、认证安全等方面的补充和延伸,GB/T 25724 则是音视频编码与打包的国家标准,也就是我们常说的 SVAC 标准。

1. GB/T 28181

GB/T 28181 是公安部发布的安防领域的视频监控互联互通标准(见图 23－90),也是目前行业内影响力最大的视频领域的互联互通标准,其全称是《安全防范视频监控联网系统信息传输、交换、控制技术要求》。GB/T 28181 目前已发布了 2011 和 2016 两个正式版本,并且在中间还有一些小的迭代更新。2016 版本中最主要的改进就是增加了基于 UDP 的 RTCP 和 TCP 的传输方式,其中 TCP 方式又同时支持主叫与被叫两种连接方式,即底层设备/平台主动向上级平台建立 TCP 连接和上级平台主动向底层设备/平台建立 TCP 连接这两种形式。

GB/T 28181 是基于 SIP、SDP、RTP 等应用层协议和 PS 视频封装方式的会话层协议,并

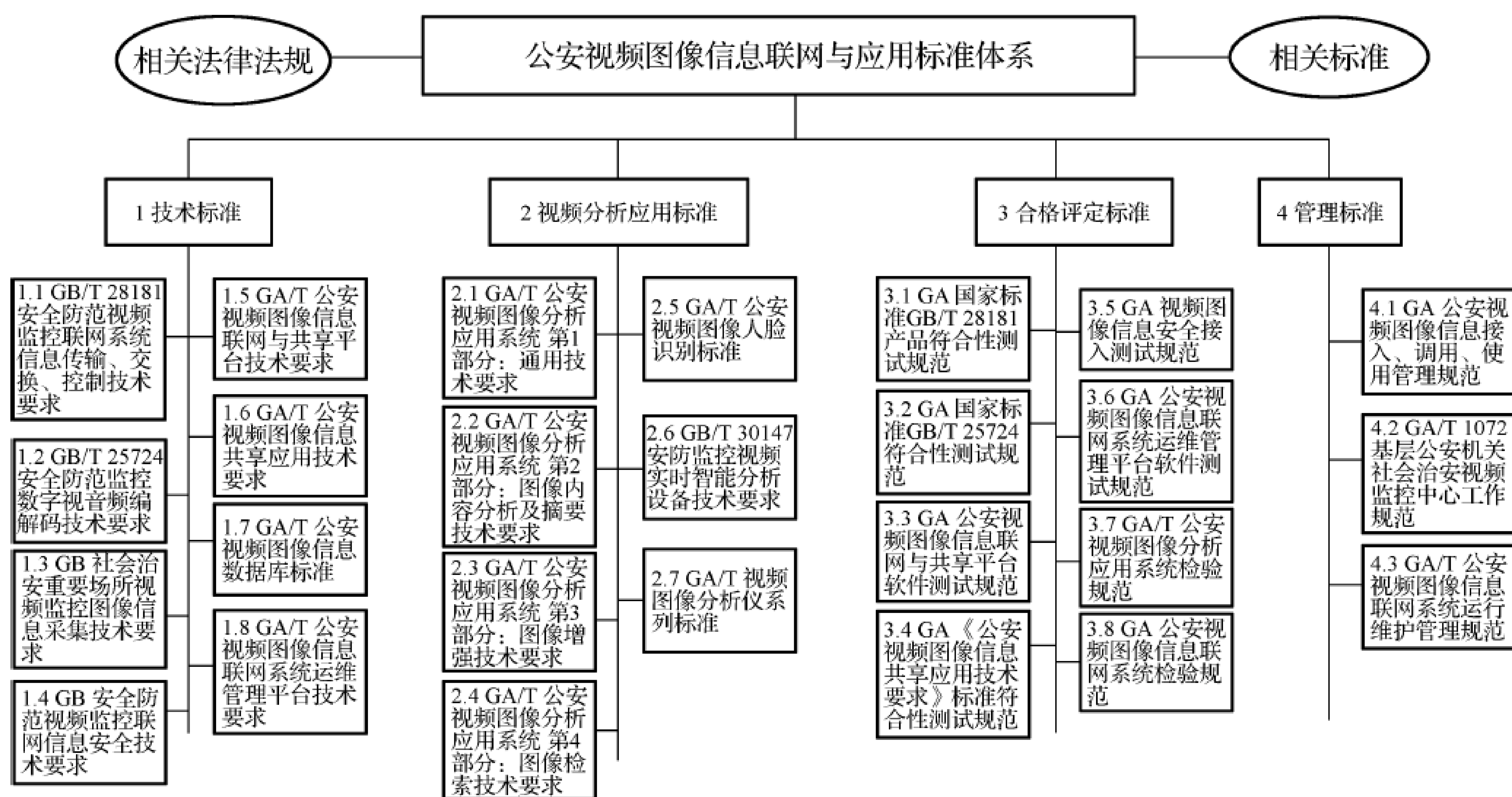


图 23－90　较早版本的公安视频图像信息联网与应用标准体系

（图片来自《公安视频图像信息联网与应用标准体系表》）

支持 SVAC、H.264 等视频编码方式和 SVAC、G711 等音频编码方式，既适用于监控终端设备接入，也适用于平台之间互联互通，并且在一定的场景下支持私网穿透特性，是一套实用性很强的标准规范。其主要包括注册与注销（见图 23－91）、心跳保活、实时视频查看（见图 23－92）、历史视频查询与查看、云台控制、报警与订阅等功能，涵盖了视频联网监控平台的大多数功能。

21.82.3.2	21.82.8.2	SIP	651 Request: REGISTER sip:65400300002000000001@21.82.8.2:5060 (1 binding) |
21.82.8.2	21.82.3.2	SIP	420 Status: 401 Unauthorized |
21.82.3.2	21.82.8.2	SIP	728 Request: REGISTER sip:65400300002000000001@21.82.8.2:5060 (1 binding) |
21.82.8.2	21.82.3.2	SIP	441 Status: 200 OK (1 binding) |

图 23－91　GB/T 28181 的注册报文

10.45.154.91	10.45.154.98	SIP/SDP	787 Request: INVITE sip:65020000001310000533@10.45.154.98:5060 |
10.45.154.98	10.45.154.91	SIP	369 Status: 100 Trying |
10.45.154.98	10.45.154.91	SIP	454 Status: 101 Dialog Establishement |
10.45.154.98	10.45.154.91	SIP/SDP	658 Status: 200 OK |
10.45.154.91	10.45.154.98	SIP	372 Request: ACK sip:65020000001310000533@10.45.154.98:5060 |
10.45.154.91	10.45.154.98	SIP	372 Request: BYE sip:65020000001310000533@10.45.154.98:5060 |
10.45.154.98	10.45.154.91	SIP	363 Status: 200 OK |

图 23－92　GB/T 28181 的视频申请与结束报文

GB/T 28181 协商信令是基于 SIP 的，即构筑在 RFC 3261 和 RFC 3265 两套标准之上，并完整保留了 SIP 的状态机和定时器等机制。例如图 23－92 中正常申请视频的流程是这样的：

（1）客户端发送 INVITE 消息给服务端请求视频，INVITE 消息中包含了所请求视频的 ID 等信息，此时客户端和服务端的 SIP 状态为临时会话建立中。

（2）服务端立即回复 100 Trying 消息给客户端，这一般是由服务端的 SIP 状态机回复的（不需要经过上层应用业务），以安抚客户端不需要重发 INVITE 报文了，此时双方的 SIP 状态机中的临时会话创建成功。

这里要解释一下,SIP 状态机也存在定时器机制。客户端在发送了 INVITE 请求而服务端没有回复时会隔一段时间再次发送 INVITE,这个时间间隔会呈倍数增加(1s,2s,4s,8s……)。其实在某些情况下服务端只是因为太忙了(例如准备流媒体相关资源)才没有回复的,并不是没收到,如果客户端不识趣地添乱拼命发会使服务端更加忙碌。100 Trying 报文就是为避免这种情况而产生的,它的含义是“我收到了,不要再发了”,它的产生者是服务端的 SIP 状态机而非更上层的业务应用,这就有效避免了忙中添乱情况的发生。

INVITE 和 100 Trying 消息的一个来回只能生成临时会话,正式会话要得到服务端业务层的“首肯”才能建立,因此临时会话只是个“草签”。

(3) 服务端准备好了媒体资源后经过 SIP 状态机向客户端回复 200 OK 报文,这个报文中也承载了服务端的流媒体参数。此时服务端与客户端状态机中的 SIP 会话都升级为正式会话,临时会话状态结束。

(4) 最后客户端要向服务端发送 ACK 报文,以表示自己收到了 200 OK 报文,也接受服务端的媒体参数,此后便可以交换媒体数据了。

GB/T 28181 应用了 SIP 中的 REGISTER、MESAGE、INVITE、ACK、BYE 等报文类型;在流媒体协商时采用 SDP(流媒体描述协议);在流媒体封装和传输方面则是采用 PS 的打包方式,且 2011 版只支持 RTP over UDP 的传输方式,同时也不兼容 RTCP 心跳机制。

2. GB 35114

GB 35114 的全称是《公共安全视频监控联网信息安全技术要求》,是公安部针对数据安全、网络安全两个主题而对 GB/T 28181 做的扩展和加强。GB 35114 在协议上全面兼容 GB/T 28181(2016 版本),也就是说仍然支持 SIP、SDP、RTP/RTCP 等几种应用层协议及其连接方式,互联互通业务方面的功能是基本不变的。

GB 35114 将安全级别分为了依次加强的 A、B、C 三个等级,其中 A 级只要求设备/平台间的信令具备身份认证功能;B 级要求在实现 A 级安全要求的基础上实现数据防篡改(数字证书)和完整性校验功能;C 级则要求在 B 级安全要求的基础上实现音视频数据内容的加密保护。

1) 先验知识

在介绍具体规范之前我们先来看一些术语,见表 23-14。

表 23-14　GB 35114 中的术语及其解释

术语	术语解释
对称密钥 (私钥加密)	定义:信息的发送端和接收端共用一个密钥,加解密速度快,安全性不如非对称密钥
	用途:功能管理平台内功能实体的签名公私钥、加密公私钥、FWSF(具有安全功能的前端设备)签名公私钥、有安全功能的用户终端签名公私钥
非对称密钥 (公钥密钥加密)	定义:密钥有一对,即公钥和私钥,公钥可发给任何请求者,私钥只有一方有,使用其中一个进行加密,只能用另一个解密。一般都是用公钥加密,私钥解密
	用途:视频密钥加密密钥、视频加密密钥,其中视频密钥加密密钥用于平台发起、变化以及对视频密钥加密

续表 23－14

术语	术语解释
CRL	证书撤销列表
ECB	电码本模式
FDWSF	具有安全功能的前端设备
IV	初始化向量
OFB	输出反馈模式
SHA	散列安全算法
VEK	视频加密密钥
VKEK	视频密钥加密密钥
VETK	视频导出传输密钥

表 23－15 中列举了规范中用到的加密算法、数字证书、资源认证、密钥管理和视频封装等方面的技术要求。

表 23－15　GB 35114 中的技术要求

技术点	技术分类	技术/规范名称	用途和其他	备注
加密算法	非对称加密算法	SM2 椭圆曲线公钥密码算法	用户身份认证，前端设备认证，服务器设备认证，平台间认证、数字签名、密钥协商	应全部采用获得国家密码管理行政机构批准的安全密码产品实现，网上基本上有开源的实现（标准 C 代码）
	对称加密算法	SM4 算法 ECB 模式	密钥协商数据加密	
	对称加密算法	SM1/SM4 分组密码算法 OFB 模式	视频加密（SM1 算法不公开，需要通过加密芯片的接口进行调用）采用 OFB 模式时，每个加密的 GOP 使用不同的 IV	
	加密散列算法	SM3 密码散列算法	完整性校验	
	随机数生成算法		必须通过 GM/T 0005—2012 规定的方法检测，已支持	
数字证书	用户证书	GM/T 0015—2012	规定了证书格式和证书撤销列表	
	前端设备证书	GM/T 0015—2012	定义了证书撤销列表的规定	
	服务器设备证书	GM/T 0015—2012	定义了证书撤销列表的规定	
	平台证书	GM/T 0015—2012	定义了证书撤销列表的规定	
	数字证书服务协议	GM/T 0014—2012	数字证书认证系统密码协议规范	

续表 23－15

技术点	技术分类	技术/规范名称	用途和其他	备注
资源认证	用户认证	GB/T 15843.3—2008	信息技术 安全技术 实体鉴别 第3部分：采用数字签名技术的机制	
密钥管理	非对称密钥管理	GM/T 0034—2014	定义了基于 SM2 密码算法的数字证书，认证系统的密码和安全要求	
视频封装与打包	视频信息封装	GB/T 25724—2017	定义了安全参数集和认证数据的语法	

表 23－16 列出了 GB 35114 规范中对于设备模块的定义及其用途，这些设备模块对于规范中的加解密及其计算加速、密钥存储等起决定性作用，并且对于安全终端设备（IPC 等）也做了详细的安全功能描述。

表 23－16　GB 35114 中的设备要求

设备名称	设备用途
FDWSF 设备（IPC）	该设备具备安全密码模块，该模块具有唯一 ID，并可以存储 VEK
	可以对视频流进行指定算法的加密
	支持 SVAC 编码和 GB/T 25724—2017 中规定的 Nal 封装方式，以及认证 Nal 和安全参数集 Nal
	支持以前端存储方式存储视频
SM1 加密芯片	支持 SM1 对称加密算法
平台安全模块	存储所有前端的视频密钥加密密钥，保存周期满足视频保存时间要求
视频导出存储介质	存储导出的加密视频，可以对视频观看时间、观看次数、复制情况的授权做存储
SVAC 加密视频流解码库	针对加密的视频流，根据 VEK 进行解码和显示，并输出 YUV 数据

2）功能分解

在 GB 35114 规范中基本上都是采用国密算法（SM 系列）进行加密工作的，包括：

- 采用非对称加密算法（SM2 椭圆曲线公钥密码算法）进行身份认证、数字签名和密钥协商。
- 采用对称加密算法进行视频数据加密保护和密钥协商时的报文数据加密，前者一般采用 SM1 或 SM4 分组密码算法的 OFB 模式，后者一般采用 SM4 分组密码算法的 ECB 模式。
- 采用加密散列算法（SM3 密码散列算法）进行完整性校验。
- 除此之外，规范中还采用了随机数生成算法用于产生随机数。

上述加密算法中，非对称密钥主要应用于平台内签名公私钥和加密公私钥、FDWSF 签

名公私钥、具有安全功能的用户终端签名公私钥等；对称密钥则主要应用于视频加密密钥和视频密钥加密密钥。

在 GB 35114 规范中数字证书包括用户证书、前端设备（IPC 等监控设备）证书、服务器设备证书和管理平台（软件）证书 4 类。数字证书采用基于公私钥的非对称密钥加密算法进行加解密。

基于满足 GB/T 28181 规范的视频监控平台软件，我们将 GB 35114 的主要功能分解为表 23 – 17 中的几个大项和子项供读者参考。

表 23 – 17　GB 35114 规范的功能分解

功能	功能分解
统一编码规则	服务器、FDWSF、安全用户终端统一编码
	FDWSF 中的密码模块，应具有 ID
用户身份认证	用户和用户证书、ID 的对应关系管理，对所有用户都要进行认证
前端设备分级	A 级（弱）：数字证书与管理平台，进行双向身份认证
	B 级（次弱）：A 级 + 视频数据签名，可校验视频内容是否被篡改
	C 级（强）：B 级 + 视频加密
设备身份认证	管理 FDWSF 基本属性信息、ID、密码模块 ID 与设备证书的对应关系等
	对所有接入的 FDWSF 进行单向或双向设备身份认证
平台间认证	平台双向认证（上下联服务器可充当逻辑信令安全路由网关）
授权与访问控制	设备身份认证基础上，平台采用基于属性或基于角色的访问控制模型对用户授权、管理控制
	访问控制粒度包括设备安全等级和存储的视频是否加密
	能访问加密视频的用户应基于数字证书认证（播放、回放、下载、删除等）
	跨域访问，信令头域加上 Monitor-User-Identive 携带的用户身份信息
视频签名与完整性校验	B 和 C 级设备支持视频数据签名和 TCP 传输
	B 和 C 级设备支持对 I 帧和关键帧签名
	平台支持对视频数据签名存储和验证——不可抵赖性
音视频加密	C 级 FDWSF 对音视频加密
	平台可以对用户权限内的加密音频进行播放、存储、回放、下载、分发、导出等操作
	音视频加密格式
	视频导出平台应更换视频密钥加密密钥
	加密视频直接存储到存储设备

续表 23－17

功能	功能分解
设备异常管理报警	对于 FDWSF 异常情况,例如非授权处理、密码模块损坏或者丢失能及时发现
	将设备(包括 FDWSF 和非 FDWSF)异常情况(报警)等写入平台日志
安全管理	设置安全管理员、安全操作员、安全审计员三类角色
	安全管理员:系统的安全参数配置、服务器启动和停止,但不具备安全业务操作权限
	安全操作员:密钥生成和导入、备份恢复等日常操作
	安全审计员:对安全事件和各类管理、操作人员行为进行审计监督
	通过数字证书、静态口令、动态口令、生物识别等认证安全管理员、安全操作员、安全审计员的身份,认证通过才能操作
日志管理	记录各种安全操作和异常安全事件(用户认证、设备认证、密钥管理、密钥协商失败、数据加解密失败、完整性校验失败)
	平台可以获取 FDWSF 各种安全异常事件日志(设备认证失败、密钥协商失败、数据加解密失败、完整性校验失败等)
非对称密钥管理	按照 GM/T 0034—2014 管理
对称密钥管理	视频密钥加密密钥在设备注册时更新,安全传输到具有安全功能前端设备的密码模块中存储
	平台要使用安全模块保存所有前端的视频密钥加密密钥,保存周期满足视频保存时间要求
	视频密钥加密密钥更新周期不超过 1 天,视频加密密钥更新周期不超过 1 小时

23.3.2　物联网协议栈

物联网设备大多部署在室外,且通常情况下的供电来源都是电池,这要求物联网设备功能简单、计算量小、功耗低、通信方式简洁,因此物联网设备一般满足以下特征:

- **设备功能单一**:大多数物联网设备只负责单一方面的功能,例如传感和告警等。
- **操作系统简单**:大多数物联设备的操作系统是轻量级的操作系统,不需要过于复杂的管理机制,协议栈大多也经过了裁剪。这样做既能满足业务的需要,也能维持较低的计算量和功耗。
- **接入协议多样**:物联网设备的接入协议包括 Ethernet、Wi-Fi、ZigBee、Bluetooth、NB-IoT、Sigfox、RFID、4G、5G、LoRa、6LoWPAN、TSN 等。这里要明确的是,物联网的接入协议不是指设备“接入”平台的协议,而是负责物联网内设备间组网和通信的协议。一般来说可以将接入协议分为近距离通信协议(近场通信 NFC)、远距离蜂窝通信协议、远距离非蜂窝通信协议和有线通信协议 4 类,如图 23－93 所示。
- **应用协议轻量多样**:在接入协议之上的应用层协议应该满足通信量少、带宽占用低、

传输效率高的特点，因此其应用协议大多比较轻量。而且由于设备种类繁多，用于匹配设备特点的应用协议非常多样，包括 HTTP/HTTPS、XMPP、CoAP、MQTT、DDS、AMQP、WebSocket、JMS 等。

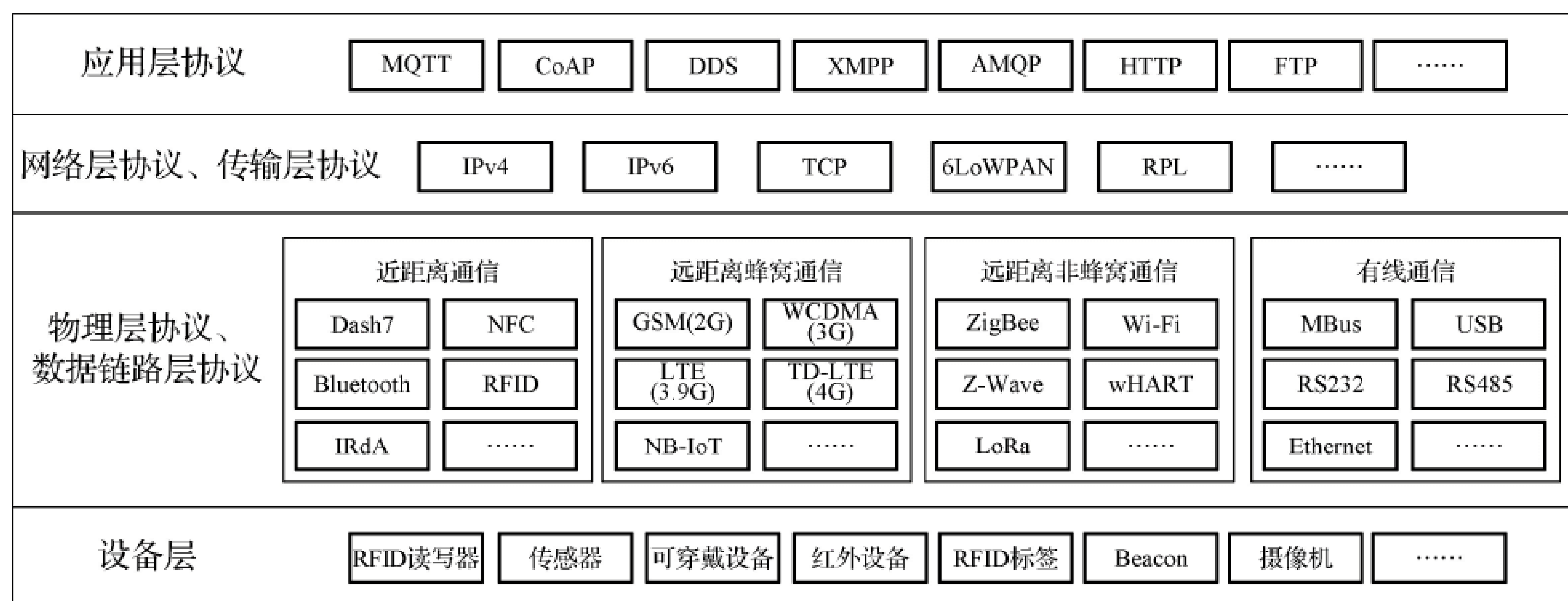


图 23－93　物联网协议分层架构(图片来自 CSDN)

从宏观意义上来说，视联网的设备(监控摄像机、视频会议设备等)也属于物联网设备的范畴，但由于监控设备规模庞大、数据传输量大、协议专一而自成一体，因此我们从更细分的角度将其划分为视联网设备。相比视联网协议，物联网协议则更加丰富多样，更加注重节能和自持力。

由于物联网领域从接入协议到应用协议都有所涉及，且应用协议都是基于接入协议的，因此我们也将物联网领域的协议称为“协议栈”。本节我们重点介绍接入协议和应用协议。

23.3.2.1　接入协议

1）Wi-Fi

Wi-Fi 是一种无线局域网技术(WLAN)，遵循的是 IEEE 802.11 标准(802.3 标准应用于 Ethernet)，是目前最为普及的一种无线组网接入技术，可以非常方便地接入互联网，当然这也要求所接入的设备必须支持 TCP/IP 协议。

2）ZigBee

ZigBee 是一种基于 IEEE 802.15.4 标准的短距离无线 Mesh 网络技术，广泛应用于工业控制和智能家居领域，具有低成本、低功耗、自组网、高安全度的特点。

ZigBee 传输速率较低，协议也很简单，因此功耗低、传输效率较高，且支持休眠和自动唤醒功能。同时由于 ZigBee 协议自带 Mesh 功能，因此可以支持 60 000 + 以上数量节点的连接，也支持鉴权认证和数据加密，并与 6LoWPAN 一样都采用了 AES 128 加密技术。

3）Bluetooth

Bluetooth(蓝牙)也是一种遵循 IEEE 802.11 标准的近距离无线通信技术，并且具有即插即用的特点，即任意一个蓝牙设备一旦搜寻到另一个蓝牙设备，马上就可以建立连接而无需用户进行任何设置。另外，Bluetooth 也具有功耗低、成本低的特点。

4）NB-IoT

NB-IoT（Narrow Band-Internet of Things）即基于蜂窝的窄带物联网，是低功耗广域网（LPWA）技术的一种，也是 3GPP 标准定义的 LPWA 解决方案。NB-IoT 具有低成本、低功耗、广覆盖等特点，多用于运营商级基于授权频谱的低速率物联网市场，特别是在位置跟踪、环境监测、智能停车、远程抄表等领域有着广泛应用。

不过 NB-IoT 的部署频率是授权的，因此必须由运营商来部署。由于 NB-IoT 构建于蜂窝网络，只消耗大约 180 KHz 的带宽，可直接部署于 GSM 网络、UMTS 网络或 LTE 网络。

5）RFID

RFID（Radio Frequency Identification）即射频识别/电子标签，这是一种非接触式的自动识别技术，可以通过射频信号自动识别目标对象并获取相关数据。

RFID 由电子标签、读写器和天线三个基本要素组成，其工作原理是：电子标签进入磁场后接收解读器发出的射频信号，凭借感应电流所获得的能量发送出存储在芯片中的产品信息（Passive Tag，无源标签或被动标签），或者主动发送某一频率的信号（Active Tag，有源标签或主动标签），解读器读取信息并解码后，送至信息系统进行相关数据处理。

6）LoRa

LoRa 是 LoRa 联盟（由 Semtech 公司发起）推出的一个基于开源的 MAC 层协议的 LPWAN 标准。严格来说，LoRa 应该是个物理层协议或者说是一种低功耗远程无线通信技术，可将其用于不同协议和不同网络架构（如 Mesh 组网、星型网络、点对点网络等）。LoRa 技术具有通信距离远、功耗低、支持多节点、成本低、抗扰能力强等特点，同时 LoRa 的传输速率较低，适合小数据传输。

LoRa 终端分为 A、B、C 三种类型，并且都支持双向通信，其分类情况如图 23－94 所示。

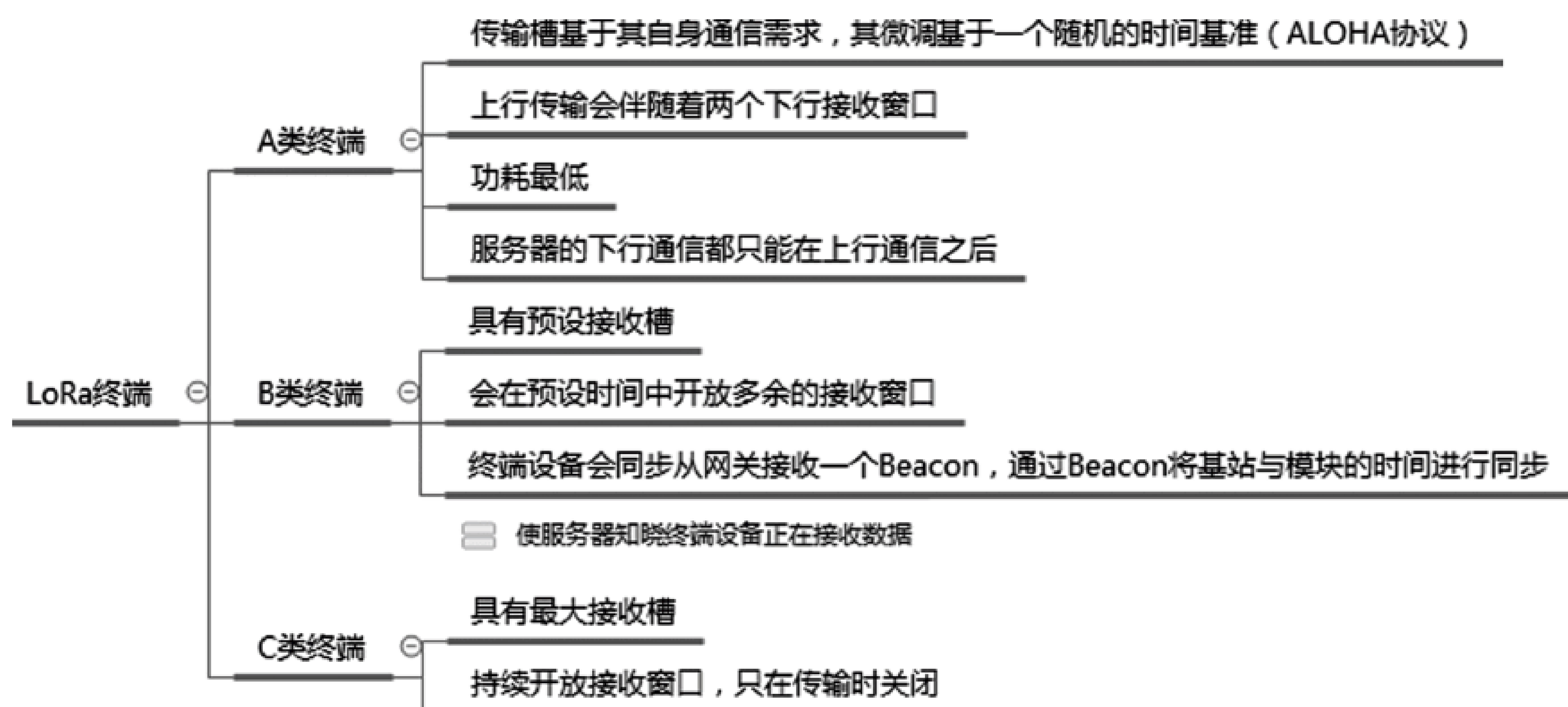


图 23－94　LoRa 终端分类

LoRaWAN 是基于 LoRa 远距离通信网络设计的一套通信协议和系统架构，如果说 LoRa 是物理层协议，那么 LoRaWAN 就是 MAC 层协议，正是由于这个原因，LoRaWAN 也被称为 LoRaMAC。LoRaWAN 是一种基于扩频技术的超远距离无线传输方案，其通信距离可以达到

15 km 以上。

LoRaWAN 分为终端、网关、服务器和应用层四大部分，其突出特点就是长距离、功耗低，并且随着 LoRa 通信速率的降低，其通信距离可以更远。LoRaWAN 也支持通信加密技术，可以保证应用层终端之间的安全。

7）6LoWPAN

6LoWPAN（IPv6 over Low-power Wireless Personal Area Networks）是一种基于 IPv6 的低速无线个域网（无线个人区域网络，WPAN）标准，即 IPv6 over IEEE 802.15.4 标准。6LoWPAN 在物理层和链路层均采用 IEEE 802.15.4 标准，但由于 IPv6 协议不能直接基于 IEEE 802.15.4 标准，因此需要在 IPv6 网络层和 IEEE 802.15.4 标准的链路层之间加入一个适配层来解决压缩、分片/重组、Mesh 路由等问题。

6LoWPAN 支持星型、树型、Mesh 网络等多种拓扑结构，其协议栈采用五层模型，即传输层之上直接为 6LoWPAN 应用层（IPv6 网络层和 IEEE 802.15.4 适配层处于第三层），如图 23－95 所示。

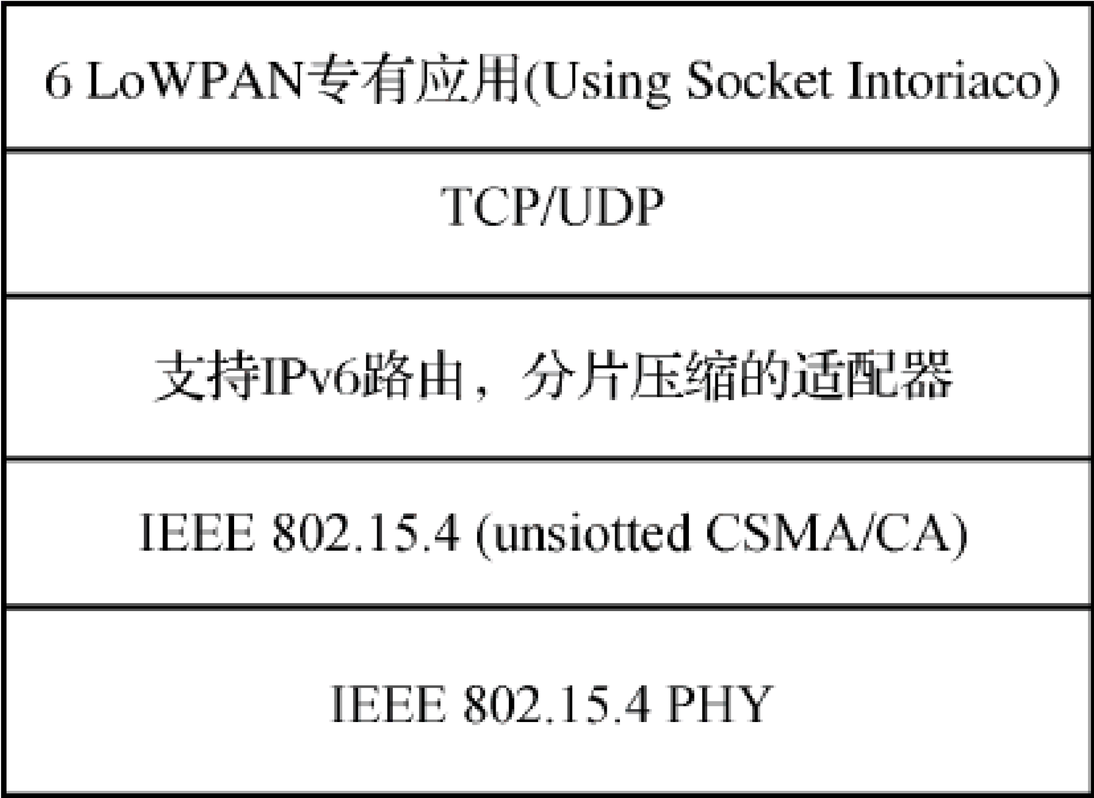


图 23－95　6LoWPAN 网络协议栈

8）TSN

TSN（Time Sensitive Networking，时间敏感型网络）是 IEEE 为在确定型以太网上传输时间敏感型数据而定义的一套标准，是下一代工业通信的标准网络。这里要注意，TSN 是在 IEEE 802.1 标准下基于特定的需求而制定的一套标准集合，意图为标准以太网建立时间敏感机制，以确保网络传输的时间确定性。由于 TSN 是 IEEE 802.1 下的标准，因此 TSN 也是链路层的标准和规范，也就是说 TSN 将会为以太网协议的 MAC 层提供一套通用的时间敏感机制，在确保以太网数据通信的时间确定性的同时，为不同协议网络之间的互操作提供可能性。TSN 在 OSI 七层参考模型中的位置如图 23－96 所示。

TSN 设计的标准非常多，但也不是贪大求全地每种都需要。例如在工业控制领域，可能用到了以下子规范：

- **802.1ASrev 时钟同步**：确保连接在网络中各个设备节点的时钟同步，并达到微秒级的精度误差。
- **802.1Qbv 时间感知调度程序**：为优先级较高的时间敏感型关键数据分配特定的时间槽，在规定的时间区间内，网络中所有节点都必须优先确保重要数据帧的通过。

	工控协议A	工控协议B	工控协议C
5-7层	负载	负载	负载
4层		TCP头	UDP头
3层		IP头	IP头
		IP封装	IP封装
2层	IEEE 802.1标准化TSN机制		
	IEEE 802.3标准化MAC		
1层	IEEE 802.3标准化PHY		

图 23－96　TSN 在 OSI 七层参考模型中的位置

➢ **802.1Qcc 网络管理和配置**：用于实现对网络参数的动态配置，以满足设备节点和数据需求的各种变化。

23.3.2.2　应用协议

物联网领域的应用协议就要简单很多了。在此我们不准备对这些协议进行详细介绍，这是因为这些协议在网上随处可见，我们只是列出这些协议的特点对比和一些简单介绍，如表 23－18 所示。

表 23－18　常用物联网应用协议比较

	DDS	MQTT	AMQP	XMPP	JMS	REST/HTTP	CoAP
抽象	Pub/Sub	Pub/Sub	Pub/Sub	NA	Pub/Sub	Request/Reply	Request/Reply
架构风格	全局数据空间	代理	P2P 或代理	NA	代理	P2P	P2P
QoS	22 种	3 种	3 种	NA	3 种	通过 TCP 保证	确认或非确认消息
互操作性	是	部分	是	NA	否	是	是
性能	100 000 msg/s/sub	1 000 msg/s/sub	1 000 msg/s/sub	NA	1 000 msg/s/sub	100 req/s	100 req/s
硬实时	是	否	否	否	否	否	否
传输层	缺省为 UDP，TCP 也支持	TCP	TCP	TCP	不指定，一般为 TCP	TCP	UDP
订阅控制	消息过滤的主题订阅	层级匹配的主题订阅	队列和消息过滤	NA	消息过滤的主题和队列订阅	N/A	支持多播地址
编码	二进制	二进制	二进制	XLM 文本	二进制	普通文本	二进制
动态发现	是	否	否	NA	否	否	是

续表 23－18

	DDS	MQTT	AMQP	XMPP	JMS	REST/HTTP	CoAP
安全性	提供方支持,一般基于 SSL 和 TLS	简单用户名/密码认证,SSL 数据加密	SASL 认证,TLS 数据加密	TLS 数据加密	提供方支持,一般基于 SSL 和 TLS,JAAS API 支持	一般基于 SSL 和 TLS	

➢ **XMPP**:可扩展通信与表示协议,这是一种基于 XML 的协议。

➢ **CoAP**:资源受限的应用协议,是一种由 IETF 制定的基于 UDP 的 REST 协议,包头为二进制压缩格式,且发送与接收可以异步实现。CoAP 用于低功耗和资源受限的网络,也重现了一些 TCP 的功能(例如可以识别需要确认的请求和无需确认的请求)。

➢ **MQTT**:消息队列遥测传输协议,这是由 IBM 制定的一种基于发布-订阅模式的二进制协议,支持 TCP 的传输方式。MQTT 由云端负责消息转发,具备完善的 QoS 机制,适合低带宽的网络情况,主要面向大型网络中的小型设备。

➢ **DDS**:面向实时系统的数据分布服务,这是一种基于发布-订阅模式的应用协议。

➢ **AMQP**:高级消息队列协议,这也是一种基于发布-订阅模式的应用协议。

➢ **JMS**:Java 消息服务(Java Message Service)应用程序接口,这是一个 Java 平台中面向消息中间件(MOM)的 API。

➢ **REST/HTTP**:REST 是一种架构设计规范,是基于 HTTP 协议的,本质上是一种 API 规范。

此外还有 WebSocket,这是一种全双工的通信组件。在 WebSocket API 中,浏览器和服务器只需要完成一次握手就可以直接创建持久性连接,并进行双向数据传输。

23.4　网络安全

广义的网络安全是个很大的领域,从防御对象的侧重点和防御技术的特征等角度可以分为以下几个细分领域:

➢ **操作系统安全**:要解决的是操作系统漏洞检测与防御、内存与缓存数据安全、CPU 侧信道攻击与防御、主机流量安全等问题。多采用挂钩技术或过滤型驱动的方式进行防御与检测。随着攻击技术的发展,这个领域的技术交锋越来越向底层渗透。

➢ **应用安全**:主要是指操作系统中应用软件的安全,包括这些软件的漏洞扫描和修复、代码安全、Web 应用安全、域名安全、防篡改、防病毒、WAF(Web Application Firewall,Web 应用防护系统)等。

➢ **访问授权安全**:包括数字证书、身份认证、数据加密等内容,着重解决的是数据资源的访问权限问题,要做到不该访问的不能访问,即使访问了也看不懂。

➢ **狭义网络安全**:着重解决各种网络攻击,例如各种 DDoS 攻击、中间人攻击、劫持攻击

等，一般采用防火墙、IDS、IPS 等技术或设备进行防御和阻断。

- **数据安全**：要解决的是数据防泄密、磁盘数据加解密、大数据安全、数据库安全等问题，一般采用各种对称或非对称加解密算法以及数字证书等技术予以实现。
- **移动安全**：这主要是指手机端的安全，包括移动端操作系统安全、移动端应用安全、移动端数据安全和访问安全等。
- **云安全**：要解决的是云环境下的网络安全、系统安全、应用安全、容器安全等问题。云安全的核心要素包括云抗 DDoS（分布式拒绝服务）攻击、云 WAF、身份验证管理、主机和操作系统安全、虚拟机安全、容器安全等内容。
- **视联网/物联网安全**：视联网和物联网安全除了要解决狭义网络安全所面临的各种攻击外，还要解决终端设备的仿冒与可信、终端设备自身的防护（防止被劫持成为"肉鸡"）、设备的安全启动/可信启动以及终端设备的安全接入等问题。
- **工业互联网安全**：这主要是指工业领域的系统安全，包括芯片、设备、操作系统、关键软件（例如工业领域常见的 SCADA 数据采集与监控系统等）、关键模块的自主研发和安全可控，保证系统不被植入恶意程序和间谍软件等 APT 代码。
- **可信计算**：目前可信计算技术已经演进到了 3.0 的时代，其主要技术特征是基于主动免疫技术识别"自我"和"非我"。
  - 模拟了人体免疫系统的工作机制，对系统"自身固有"和"外来移植"的成分进行鉴别，从而破坏和排斥进入系统机体内的有害成分（APT 等恶意程序）。
  - 相当于为网络和信息系统培育了免疫能力，使其自身就具有防护的能力，其主要技术手段是以密码为基础的身份识别、状态度量、保密存储等。
- **自主可控**：由于众所周知的原因，自主可控已经成为我国网络安全领域的重要一环。自主可控主要是指芯片、主板、操作系统和关键软件/中间件的自主可控，特别是核心技术、核心模块的自主可控。要解决好在目前的软件生态下芯片/操作系统可移植性的问题，例如从 X86/X64 架构向 ARM 架构的切换、从 Windows 系统向国产 ARM/X86/X64 架构操作系统的切换、基于新操作系统的编译器研发、虚拟机的研发等。

接下来主要讲述狭义的网络安全问题都包含哪些方面，并着重介绍视联网领域的安全问题与解决思路，最后简单总结自主可控的相关现状。

23.4.1　狭义网络安全

狭义网络安全是指设备在网络上所遭受到的各种攻击，例如 ICMP Flood 攻击、SYN 攻击、中间人攻击、DNS 劫持等。下文中我们将列举一些常见的网络攻击类型。

1）SYN 攻击

SYN 攻击是一种 TCP Flood 攻击，利用的是 TCP 三次握手过程中的漏洞。三次握手的正常过程如图 23－97 所示。



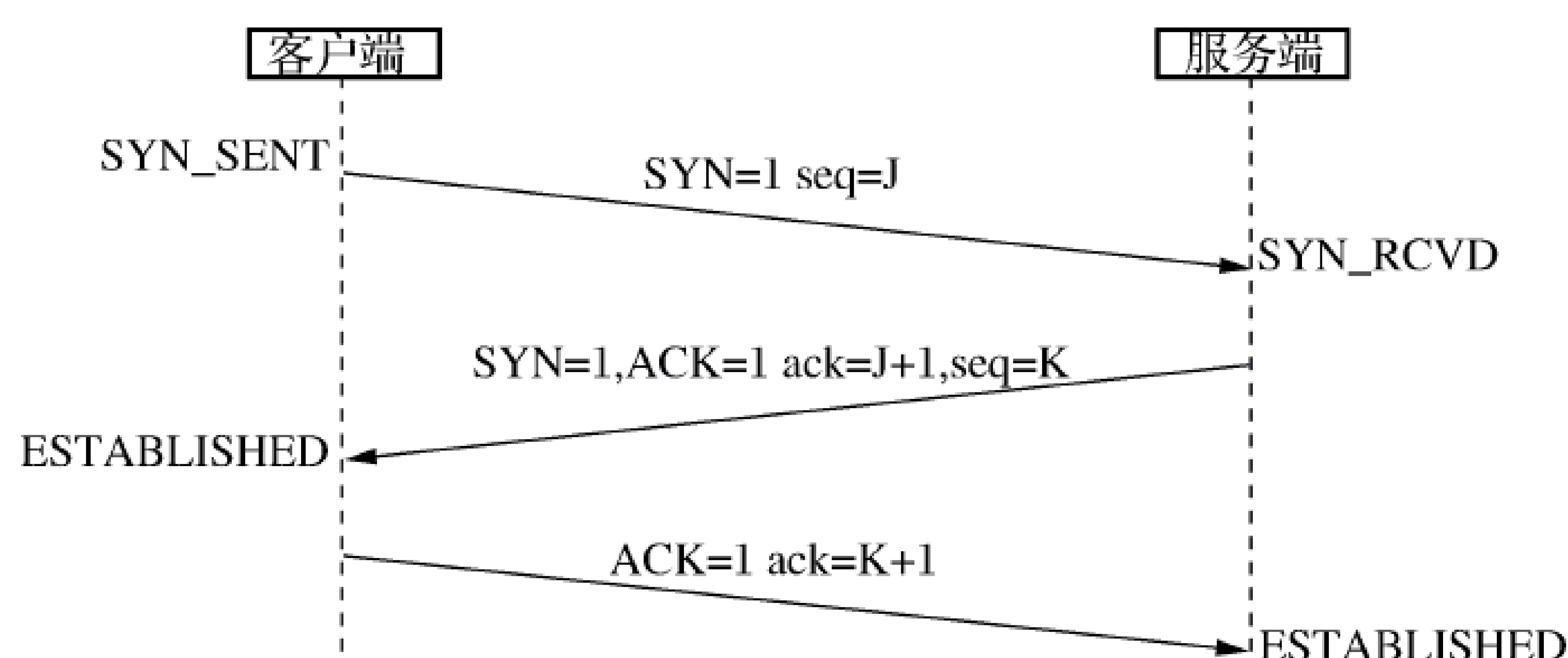


图 23-97　TCP 的三次握手过程

SYN 攻击就是利用三次握手过程中的第二步——服务端发送了 SYN-ACK 之后、但在收到第三步的 ACK 之前的半连接状态(服务端处于 SYN_RCVD)进行的 Flood 攻击。其具体做法是伪造若干不存在的地址向服务端大量发送 SYN 包,使服务端的 socket 产生大量半连接状态。由于源地址不存在,因此服务端回复的 SYN-ACK 不会被确认并直到连接超时。在这种情况下大量服务端的 SYN 包将长时间占用未连接队列,从而导致正常的 SYN 请求因为队列满而被丢弃。因此 SYN 攻击也是一种 DoS(拒绝服务)攻击。

一般来说,对于 SYN 攻击的防御可以采用以下两种策略:

➢ **过滤网关防护**:这些网关是指防火墙等设备。具体可以采用如下机制抵御 SYN Flood 攻击:

- **设置网关超时机制**:防火墙收到客户端的 SYN 请求后开始计数,当计数器到期后还未收到客户端的 ACK 包,则向服务端发送 RST 包以结束当前的半连接状态。这是一种主动缩短半连接等待周期的机制。
- **SYN 网关机制**:网关收到客户端的 SYN 包后转发给服务端,在收到服务端的 SYN-ACK 回复后也转发给客户端。此时再以客户端的名义给服务端发送 ACK 包以使服务端尽快结束半连接状态而进入 ESTABLISHED 状态。当客户端的 ACK 包到达时,若包中有数据则转发给服务端,没有则丢弃。这是一种避免半连接等待的机制。
- **SYN 代理机制**:SYN 代理网关加装在服务端之前,由代理网关去阻挡 SYN Flood 攻击,这要求代理网关有很强的防范和处理能力。有的 SYN 代理具备 Cookie 源认证功能,当代理网关响应 SYN-ACK 报文时带上特定的 Cookie(不是指客户端起始 seq num,而是一个随机的 seq num,因为双方的起始报文序号可以是任意的),如果是真实的客户端则会返回 ACK(seq num 加 1)或 RST(seq num 不正确),而恶意的客户端则不会响应返回,此时可以利用黑白名单机制将这些源地址作区分标注。

➢ **加固 TCP/IP 协议栈**:通过在 TCP/IP 协议栈驱动中增加防御机制而使服务端本身具备一定的防御能力。可以对 TCP/IP 协议栈驱动做以下加固:

- **延缓性机制**:诸如 SynAttackProtect 机制等,包括在服务端系统中增加最大半连接

数和缩短超时时间等措施使服务端系统能处理更多的 SYN 连接。

- **SYN Cookies 机制**：服务端在第二步回复 SYN-ACK 报文时并不分配缓冲区队列等资源，而是根据上一步的 SYN 包计算出一个 Cookie 值并作为返回的 SYN-ACK 报文的 seq num，当客户端返回 ACK 包时会根据包头信息计算出 Cookie 并与返回的确认序列号作对比，相同则证明这是一个正常的连接，之后再分配资源建立连接。

2）TCP 会话劫持攻击

由于 TCP 协议并没有对 TCP 的传输包进行身份验证，所以在我们知道一个 TCP 连接中的 SEQ 和 ACK 的信息后就可以很容易地伪造传输包，假装成任意一方与另一方通信，这一过程被称为 TCP 会话劫持（TCP Session Hijacking）。

一般来说劫持分为 5 个步骤，即选定攻击目标（支持 TCP 协议，报文明文传输等）、选定 TCP 会话、猜测序列号、迫使客户端下线、接管会话。TCP 会话劫持的难点在于猜测序列号，因为 TCP 本身的限制，ACK 序列号必须单向递增。

对于 TCP 会话劫持的防御策略是在网络层采用 IPSec 协议，从而使得劫持者无法猜测出会话序列号。

3）UDP Flood 攻击

UDP Flood 是一种带宽类的 DoS 攻击，即短时间内向目标主机发送大量 UDP 报文，一来是消耗带宽资源，二来是消耗被攻击主机的计算资源。

一般采用限流的方式抵御 UDP Flood 攻击，包括基于目的地址的限流、基于目的安全区域的限流和基于会话的限流等策略。

4）ICMP Flood 攻击

ICMP Flood 攻击的原理是短时间内向特定目标不断请求 ICMP 回应，致使目标系统负担过重而不能处理合法的传输任务。这是一种典型的 DoS 攻击。

5）IP 分片攻击

IP 分片攻击的原理是攻击者给目标主机只发送一片分片报文（IP 报文）而不发送所有的分片报文，这样被攻击者便会一直等待直到超时，从而消耗了大量计算资源。

6）滴泪攻击

滴泪攻击也是分片攻击的一种。这种攻击的原理是 IP 协议在传输过程中对过大的数据会进行分包处理，传输到目的主机后再到堆栈中进行重组。

为实现重组，IP 包的包头中含有说明该分段是源数据中哪一段的信息。如果发送伪造的含有重叠偏移信息的分段包到目标主机，当被攻击主机试图将分段包重组时，由于分段数据的错误，重组过程会引起内存错误，继而可能导致协议栈的崩溃。

7）Smurf 攻击

在这种类型的攻击中，攻击者把 ICMP ECHO 的源地址设置为一个广播地址，计算机在回复 ICMP REPLY 的时候就会以广播地址为目的地址，这样本地网络上所有的计算机都必须处理这些广播报文。如果攻击者发送的 ICMP ECHO 请求报文足够多，产生的 REPLY 广

播报文就可能把整个网络淹没。

Smurf 攻击的防范措施包括:路由器禁止 IP 广播包、使网络上所有主机禁止对目标为广播地址的 ICMP 包响应等。

以上的攻击类型都是比较常见的,另外还有死亡之 ping 攻击、IP 欺骗攻击、Land 攻击等,还包括针对路由器的路由协议攻击、针对转发设备的转发表攻击等,这里就不一一赘述了。

23.4.2 视联网安全

视联网除了要应对狭义的网络安全问题外,还要针对与视频设备相关的攻击做出防御和阻断,特别是在公安视频专网中,这些问题尤为突出。与视频相关的安全问题包括监控设备接入(私接)、监控设备仿冒、服务器安全、协议安全、数据截获、网络隔离等,而这些问题可以归纳为两大类:

- **设备接入问题**:包括设备私接、准入、仿冒、被操控等问题;
- **网络隔离问题**:包括协议安全、数据安全、视频专网与互联网的隔离等问题。

1. 设备接入问题

监控设备的接入一般是通过 GB/T 28181 或 Onvif 等协议实现的。但是这些协议都是五层及以上的协议,平台侧验证的是协议的报文头和报文体,一般不验证协议从哪来、到哪去,是不是由合法的监控终端发出的等一系列与安全相关的信息。这些问题总结起来就是私接、准入、仿冒,这里的仿冒是指以别的什么设备来代替监控终端接入到网络里。

要解决这些问题,除了在平台侧做校验、做认证之外,比较有效的技术手段就是“设备指纹”,由于视联网领域的大多数设备都是视频监控设备,因此也被称为“IPC(IP Camera)指纹”。

通俗地讲,IPC 指纹就是利用设备里面的操作系统、厂商信息、MAC 地址、IP 地址、端口号、协议类型等信息进行综合识别而形成的跟每一台具体设备相关的私有信息,这些信息像指纹一样具有设备标识的唯一性,而且不容易仿冒。特别是每种终端从第二层到第七层每一层的协议都是不同的,因此可以使用第二层到第七层的协议特征作为设备指纹的基础信息。图 23-98 展示了通过 IPC 指纹进行设备准入的流程。

设备厂商需要构筑自己的设备指纹库,例如根据上述信息对指纹的基本形态构成模板链,使用时进行匹配。而主机侧可以构筑可信的白名单库来对终端设备进行准入比对。一般来说,IPC 指纹检测可以分为主动和被动两种方式:

- **主动方式**:由主机侧向设备发送信息来获取回复报文,以获得 IPC 设备的厂商、系统版本、MAC 地址等信息。例如向某设备发送 ICMP ECHO 报文,根据返回的 ICMP REPLY 报文的报文头和报文体的信息来判断识别指纹信息。
- **被动方式**:通过监听截获设备的通信协议来研判和识别设备的指纹。

设备指纹不仅可以应用在视联网领域,在物联网领域也是可以大有作为的。但由于物

联网领域的协议比较繁杂而且轻量级，不像视联网领域的协议庞大而单一，因此若只通过通信协议本身来判断指纹，其误判率会比较高。

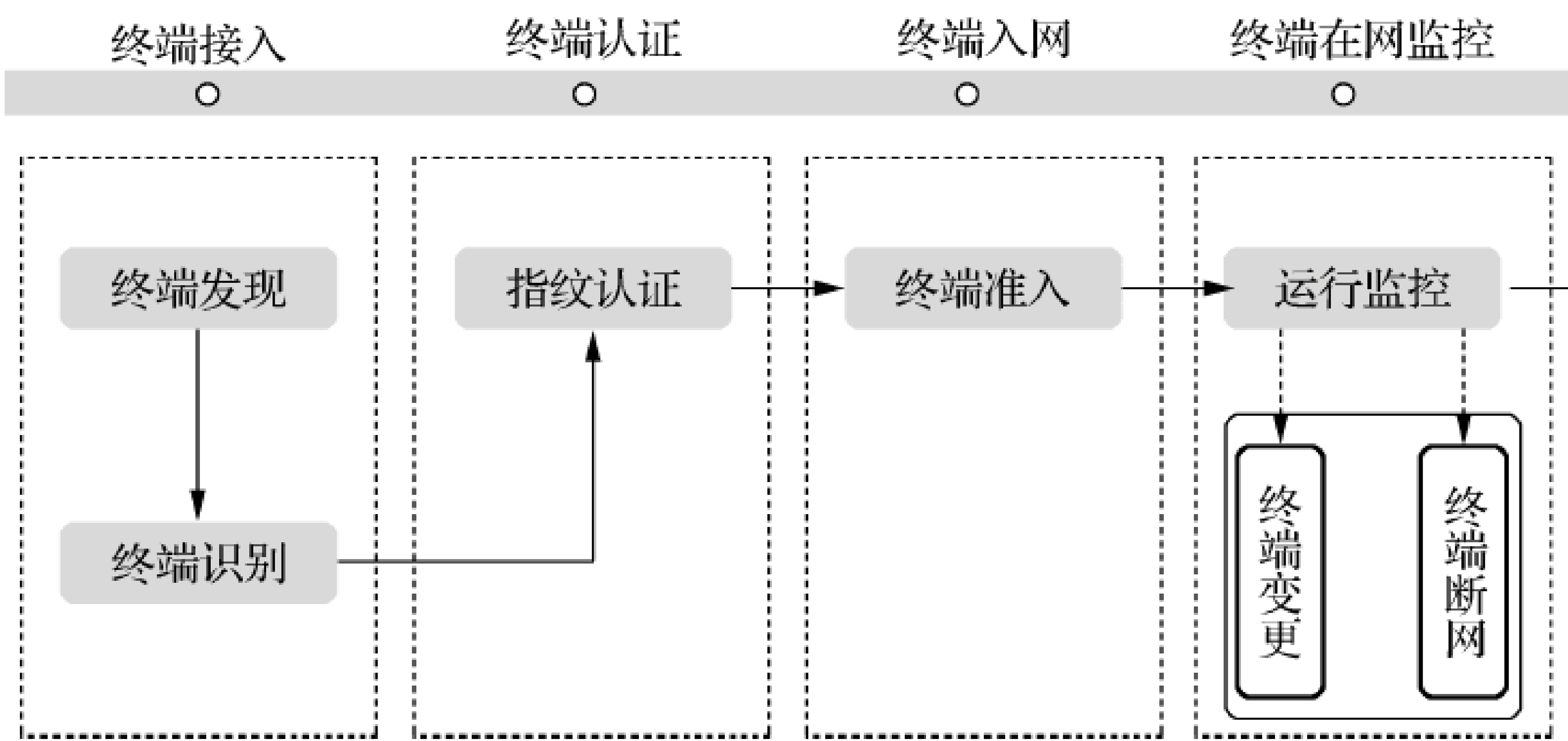


图 23－98　通过 IPC 指纹进行设备准入的流程

2. 网络隔离问题

在视联网中经常会遇到网络隔离的情况，例如办公网、政务网、公安专网/内网、互联网等各种网络的隔离。由于每种网络的安全级别不一样，面临的安全威胁也不尽相同，因此在面对多种异质网络的问题时要特别注意安全隔离，既要做到保证数据的互联互通，又要做到安全威胁的阻断隔绝。特别是公安专网/内网的安全等级很高，不允许有攻击和威胁的发生；而互联网则面向大众，其安全级别较低，网络上的攻击事件层出不穷，因此公安专网/内网和互联网之间做隔离时要格外注意这个问题。

目前可采用较多的手段对这些网络进行隔离保护，例如中转网关＋缓冲隔离网络的方式，但比较常用和制式化的方式还是网闸和安全接入平台。

1）网闸

说到网闸，就不得不先提及“安全隔离”技术。安全隔离（GAP）技术的核心是定义应用数据白名单，并通过安全方式获取数据，进行数据校验和内容检查，再通过安全方式发送数据。网闸的基本架构体现了这一核心思路，如图 23－99 所示。

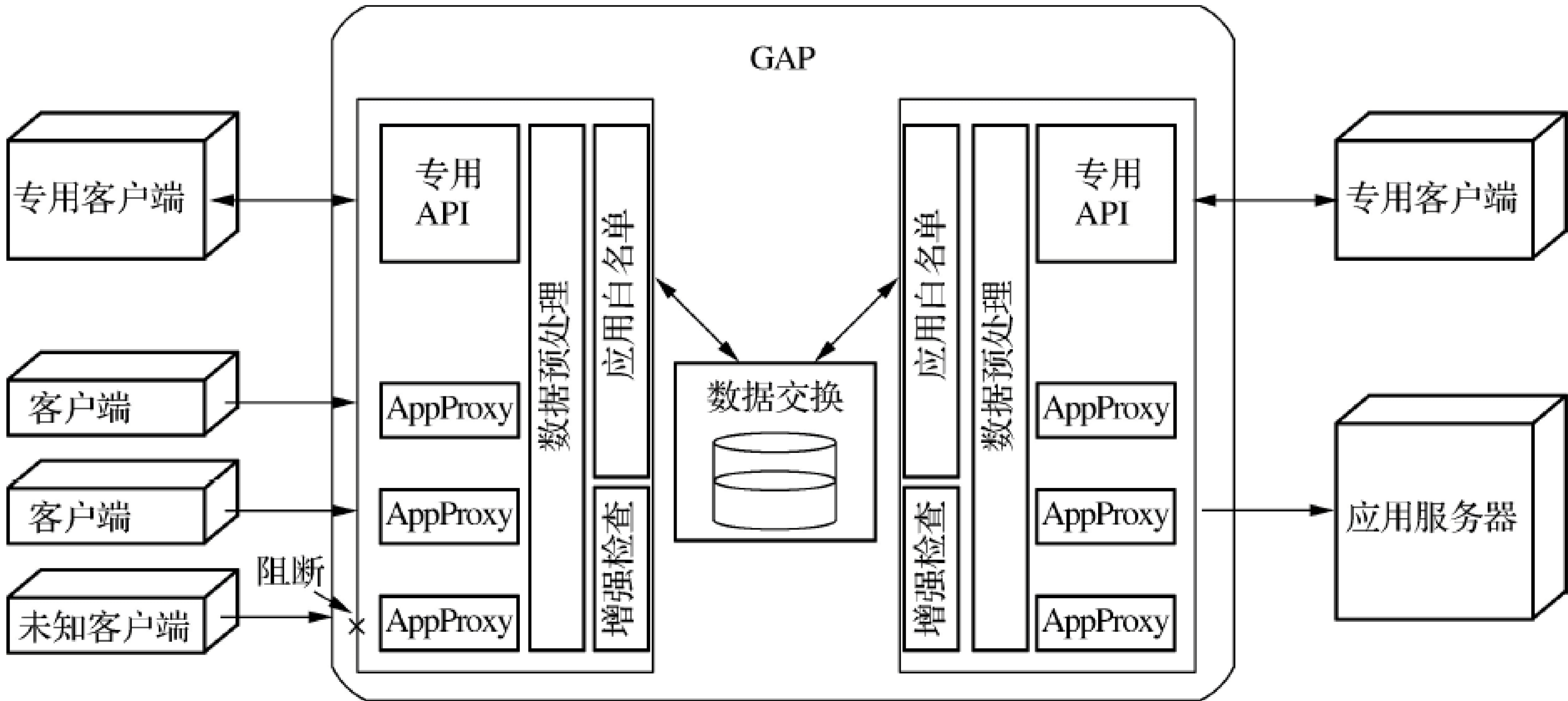


图 23－99　应用 GAP 技术的网闸体系结构

从图 23－99 中可以看出，基于 GAP 技术的网闸一般采用以下策略和机制实现隔离：

➢ 多台主机采用专用硬件串联形成纵深防御的体系架构，这样即使外侧的主机被攻击，内部的主机也是安全的。
➢ 硬件架构中多采用专用的防篡改硬件阻隔 TCP/IP 通信，以保证数据传输和检查机制的固件化特性和防篡改特性，做到真正的物理隔离。
➢ 只允许白名单内来源的请求，拒绝任何未知来源的主动式请求。
➢ 应用层数据的读写都是通过主动请求、专用 API 或专用客户端的方式进行的，这些 API 或客户端能够针对各种未知的请求进行堆栈溢出、拒绝服务攻击等安全检查；同时 GAP 以主动的方式向客户端请求资源，这样也可以避免开放端口遭受攻击。
➢ 支持用户通过扩展定义的方式对数据内容进行增强检查，包括检查内容中是否存在可执行代码、报文格式是否规范等。

从表 23－19 中可以看出，GAP 技术与防火墙技术面向的层次是相当不同的，GAP 要对应用数据进行检查，面向的是第五层到第七层的数据，而防火墙更多的是对第四层及以下层次的协议数据进行校验。

表 23－19　GAP 与防火墙技术对比

项目	安全隔离（GAP）技术	防火墙（Firewall）技术
访问控制特点	基于物理隔离的白名单控制	基于连通网络的黑名单控制
硬件特点	多主机形成纵深防御，保证隔离效果	单主机、多宿主
	专用隔离硬件	无专用数据交换硬件
软件特点	不允许 TCP 会话	允许 TCP 会话
	不允许从外到内的访问	允许从外到内的访问
安全性特点	可以最大限度防止未知攻击	不能防止未知攻击
性能与应用特点	确保安全性能所需的管理和维护工作量小；性能适中；需要与应用系统结合	需要 7×24 监控，确保安全性能；对应用透明

2）视频安全接入平台

视频安全接入平台是个非常贴近行业应用的产品。公安的视频应用资源一般划分为社会面资源和公安面资源两类。所谓社会面资源大多是非公安承建的视频资源，这些资源一般分布在企业网、互联网、园区网等安全等级相对较低的网络中；而公安面资源就是由公安承建并分布于公安视频专网的视频资源，这种网络的安全等级很高。而在公安的网络体系中还存在公安内网，这部分网络中存放有密级更高的数据资源，承载了公安日常办公、办案的信息和数据，因此对内网的访问必须慎之又慎。可以看出，这三种网络按照安全等级的高低排序为：公安内网 > 公安专网 > 社会资源网。

视频安全接入平台就是在这种场景下诞生的，它的核心任务是采用必要的安全隔离设备，对应用服务区进入公安信息通信网（公安专网/内网）的数据进行协议剥离，视频数据和

控制信令以专用数据块的方式“摆渡”传输,在此过程中还要进行必要的校验,以此来实现公安信息通信网和应用服务区之间的安全数据交换,如图 23 - 100 所示。

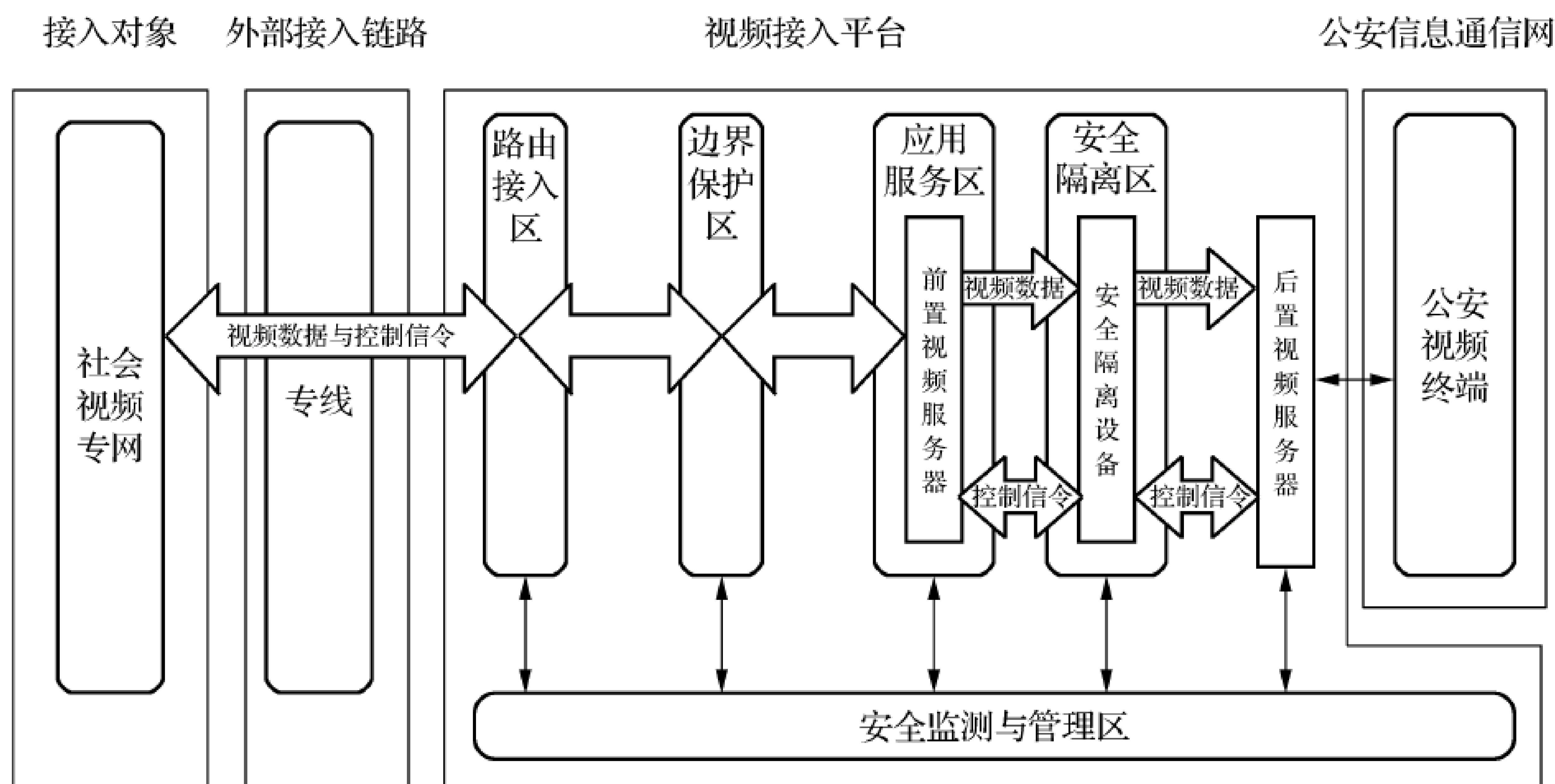


图 23 - 100　某视频安全接入平台的体系架构

视频安全接入平台从逻辑上可以分为路由接入区、边界防护区、应用服务区、安全隔离区等几个部分,其中针对视频媒体数据又可以增设前置和后置视频服务器,以方便对视频进行透传和校验。

综上所述,视频安全接入平台具有以下特征:

(1) 视频协议的安全控制,包括:

➢ 仅允许高密级网络中被授权的终端访问低密级网络的资源,不允许反向访问。
➢ 要对交互的信令报文与媒体数据进行校验,例如报文头的源和目的端的 URI 是否合法、源和目的端的地址和端口是否合法、封装媒体流的 RTP 包头中的 SSRC 是否合法等。
➢ 安全级别可以配置,可以动态调整对数据包的检测深度。

(2) 通信端口支持动态开启和关闭,包括:

➢ 视频安全接入平台只允许开放少量的端口(TCP + UDP),从而杜绝由于端口开放过多、时间过长而造成的风险。
➢ 不交换数据时流媒体的通信端口处于关闭状态,只有需要时才启用,使用完成仍然关闭。

(3) 身份认证与授权:视频安全接入平台支持使用人员的身份认证(例如通过数字证书的方式认证)、用户访问授权以及权限管理等功能。

(4) 恶意代码防护,包括:

➢ 安全接入平台禁止来自外部的对内网操作系统、文件资源或其他应用进程的操作指令。

- 必要时对视频流的内容进行校验，检查是否“夹带”可执行文件数据，同时也要求视频流的接收端安装必要的防护软件。

（5）访问控制：视频安全接入平台能够细粒度地对访问进行控制，例如校验访问的源和目的端的地址和端口、访问时间、行为权限等。

总之，视频安全接入平台是个非常贴近视频监控业务应用的产品，其安全级别能动态调整设置，支持目前较为通用的规范标准（GB/T 28181、GB 35114 等），具备对视频流进行深度包检测（Deep Packet Inspection，DPI）的能力，并在此基础上保证一定的通信效率。视频安全接入平台更像一个“边检站”，对收到的所有数据都会掂量掂量是否合法。除此之外，还有光闸、二维码网关等安全设备都可应用于视联网安全。

23.4.3 自主可控

近几年来，自主可控、安全可信越来越为业界所认同。在信息技术领域，在两个方面有着较高的自主可控的呼声，一个是芯片，另一个是操作系统。芯片方面最核心的还是地位最高且复杂度也最高的 CPU，而操作系统方面最核心的是与现有 X86/X64 架构兼容匹配的桌面系统。

CPU 的自主可控的标准是什么？目前国内比较公认的就是三条：

- CPU 指令系统可持续性的自主发展；
- CPU 核心源代码的自主开发；
- CPU 研制厂商符合安全自主保密的可靠性要求。

从这个角度来说，CPU 的自主可控之路貌似只有两条，即：

- 自己定义全新的指令集，重新打鼓另开张；
- 购买 CPU 指令集授权（例如永久性授权），并在此基础上扩展自己的指令集。

可以看出，前者是一条“重新发明轮子”之路，这条路的确是最符合“自主可控、安全可信”的，但却无法适应于当前主流处理器架构下成熟的软硬件生态，更是直接地将现有的软硬件体系当作自己的对手盘，因此无论是可行性还是时间等成本都是无法估量的，是不可取的。

而后者才是目前唯一可行、唯一现实的道路，也是为国内大多数 CPU 厂商所采用的。这条路相比软核授权和硬核授权对厂商的要求更高，因为架构授权相当于只拿到了处理器的接口描述和设计思想，布线和内部接口等工作都要自主设计，难度可想而知，当然自主化程度也是最高的。

既然谈到了指令集授权，我们不得不先重温一下 CPU 的架构流派（指令集架构在前文中已有所描述）。

- **X86/X64 体系架构**：是服务器和桌面系统中占有率冠绝群雄的体系架构，天津海光和上海兆芯是国内生产 X86 CPU 的龙头企业。但致命的缺点是 X86 指令集是不开放授权的，因此在商用领域 X86 CPU 的自主可控是个很尴尬的话题。




- **ARM 体系架构**：ARM 架构一个最突出的特点是节能，包括飞腾、华为海思、展讯、华芯通等厂商都在从事 ARM CPU 的研发，其中飞腾可以对 ARM CPU 计算模块的代码进行修改；华为拥有 ARMv8（64 位指令集）的永久性授权，并且出厂的泰山服务器都已开始采用 ARM CPU 了。不过在商业市场环境下 ARM 指令集的扩展只能在预留的接口下实现。
- **MIPS 体系架构**：包括国内的龙芯、君正在内的 CPU 均采用 MIPS 架构，其中龙芯获得了 MIPS 架构的永久授权，可以在指令集上做修改。MIPS 架构在交换机、工控机等特种计算机领域有较多应用。
- **Alpha 体系架构**：包括国内的申威等品牌在内的 CPU 采用了以高性能著称的 Alpha 架构，其中申威获得了 Alpha 架构的永久授权，可以在该体系架构下扩展指令集。
- **Power 体系架构**：也是一种高性能架构，国内的中晟宏芯 CPU 采用的就是这种架构。

除以上体系外，在精简指令集领域还有一个新秀，即 RISC-V。这是一种免费、开放的处理器架构，支持自由扩展而没有授权方面的限制，并且支持 32 位和 64 位两种寻址体系，性能方面也很不错，似乎是自主可控 CPU 的另一片曙光。

不过，虽然 RISC-V 对在核心指令集之外的扩展没有任何限制，但如果任由这种百花齐放的格局形成也会造成 CPU 生态的碎片化。对于不被纳入 RISC-V 核心标准的扩展，其生态不被支持，这样就需要自己打造生态，可能会遭遇适配、专利等各种问题。

综上所述，目前复杂指令集（CISC）体系还是以 X86/X64 称王，精简指令集（RISC）体系则以 ARM 为尊，并且随着软件定义和虚拟化技术的发展，通用型 CPU 必然会占据更多的市场份额，因此可以预见以 X86/X64 和 ARM 为代表的 CPU 体系架构会逐渐一统江湖。

除了 CPU，操作系统的自主可控同样重要。不过这个领域的情况似乎要比 CPU 领域好一些，目前国产操作系统大致可以分为如图 23－101 所示的几种。

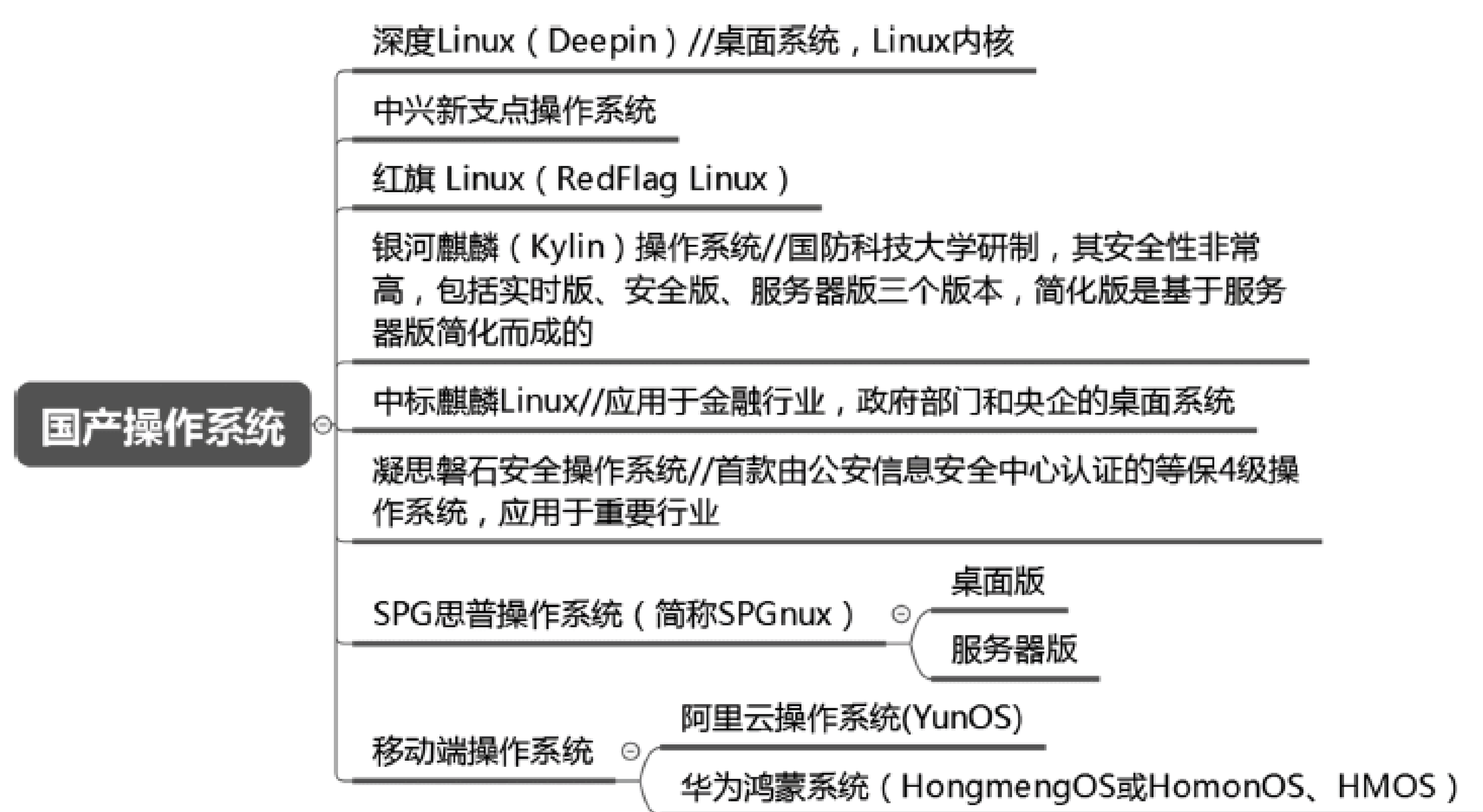


图 23－101　国产操作系统

以深度操作系统为例，它又可分为桌面版、服务器版（与桌面版操作系统相比，服务器版一般来说内核配置参数不同：桌面版具有更高的前台响应性能，服务器版则更倾向于后台服

务;在电源管理方面,桌面版配置更节能,服务器版配置则可能启用高性能,功耗更大)、安全操作系统、虚拟化系统和专用定制系统五大类。

➢ **桌面版**:不但支持 X86/X64、ARM 架构的处理器,也支持龙芯、申威、飞腾等非 X86 和 ARM 架构的处理器,并且基于 deepin-wine 技术支持运行大量的 Windows 软件。

➢ **服务器版**:支持 X86/X64 架构处理器和部分国产处理器平台,同时也支持运行 Docker + K8s、OpenStack + KVM 等应用场景。

➢ **安全操作系统**:该版本符合公安部信息安全等保标准,适用于政企单位的机要部门,支持国际标准的 TPM 和国内的 TCM/TPCM 标准,并具有以下特性:

- 支持高强度的口令认证以增强用户身份鉴别的安全性;
- 可根据文件属主确定自主权限,做到自主访问控制;
- 对主体和客体进行分级安全标记;
- 支持管理分权,杜绝控制权限泛滥引起的安全风险;
- 支持全登录周期的安全审计、警告和追踪;
- 支持用户数据完整性,阻断客体攻击;
- 支持数据保密性策略,确保信息安全;
- 支持限定用户资源;
- 支持半自动化的敏感信息发现;
- 提供配置、散列、密钥、不可读写、只可校验等安全机制;
- 支持可信度量,从 BIOS 起就对设备、内核和文件进行可信性度量。

➢ **虚拟化系统**:基于 VOI(Virtual OS Infrastructure,虚拟操作系统基础架构)技术的虚拟终端管理系统,支持通过网络而非本地硬盘读取虚拟安全系统镜像到本地以启动操作系统。

➢ **专用定制系统**:可针对军工、金融、轨道交通等行业进行量身打造。

当然,作为与 CPU 同等重要的操作系统,研发出来只是解决了有没有的问题,生态兼不兼容才是最核心的好不好的问题。国产操作系统特别是桌面和服务器系统在这一点上任重而道远。近年来,中电集团也是花大力打造了中国的“PK 体系”,“P”“K”两个字母分别代表了天津飞腾(Phytium)和银河麒麟(Kylin),前者是 ARM 架构 CPU 的提供商,后者是国产操作系统的开发者,二者互相磨合互相适配,打造中国的“Wintel”生态体系。

本章小结

在本章中,我们将 OSI 七层参考模型与 TCP/IP 五层模型作了对比,并明确了应用层协议的功能和范围。在协议栈的每一层我们都挑出来了一些有代表性的或比较新颖的协议进行介绍。这些协议都是与应用软件密切相关的。

网络层协议最主要的功能一个是数据封装与传输,一个是传输控制和差错控制。

在传输层,我们引入了 RUDP 的概念。RUDP 本质上就是 UDP 的快速传输 + TCP 的拥塞控制,中和了两种传统协议的优点以支持快速可靠的连接。本章要介绍的 QUIC、UDT 等协议都是 RUDP 的具体实例和应用。在本章中我们也着重讲述了 TCP 拥塞控制机制及其相关算法,特别是在长肥网络中的拥塞控制。但是,拥塞控制不是传输技术,而是一种控制理论。

在传输层协议部分,重点介绍了 SCTP、QUIC、UDT、TLS 协议以及 TCP 的拥塞控制和加速技术。

在应用层协议部分,分成视联网协议栈和物联网协议栈两部分予以介绍。在视联网协议栈部分介绍了:专门用于视频传输的 SRT 协议,RTP/RTCP 的加密版本 SRTP/SRTCP 协议,音视频封装协议 PES、TS、PS,视频会议相关协议 H.323 协议簇,以及最上层的视频互联互通协议 GB/T 28181 等。在物联网协议部分介绍了什么是接入协议,什么是应用协议。这两种协议一下一上,也构成了协议栈的层次结构。

最后,在网络安全部分分别介绍了:狭义的网络安全包括了哪些内容,面临着哪些威胁;视联网安全是怎么回事,特别是在公安专网环境下怎样做到多源异质网络的融合;自主可控包含哪几方面,CPU 和操作系统自主可控的现状是怎么样的。

参考文献

[1] 毛德操. Windows 内核情景分析:采用开源代码 ReactOS[M]. 北京:电子工业出版社,2009.

[2] 潘爱民. Windows 内核原理与实现[M]. 北京:电子工业出版社,2010.

[3] X64 Deep Dive [OL]. [2019-01-26], http://www.codemachine.com/article_x64deepdive.html.

[4] RUSSINOVICH M, SOLOMON D, IONESCU A. 深入解析 Windows 操作系统[M]. 6 版. 潘爱民,等译. 北京:电子工业出版社,2014.

[5] 张帆,史彩成,等. Windows 驱动开发技术详解[M]. 北京:电子工业出版社,2008.

[6] 谭文,陈铭霖,等. Windows 内核安全与驱动开发[M]. 北京:电子工业出版社,2015.

[7] 谭文,杨潇,邵坚磊,等. 寒江独钓:Windows 内核安全编程[M]. 北京:电子工业出版社,2009.

[8] 张佩,马勇,董鉴源,等. 竹林蹊径:深入浅出 Windows 驱动开发[M]. 北京:电子工业出版社,2011.

[9] 戚利. Windows PE 权威指南[M]. 北京:机械工业出版社,2011.

[10] 埃里林 C C,戴维斯 M,伯德莫 S,等. 黑客大曝光:恶意软件和 Rootkit 安全[M]. 姚军,等译. 北京:机械工业出版社,2011.

[11] HOGLUND G, BUTLER J. Rootkits:Windows 内核的安全防护[M]. 韩智文,译. 北京:清华大学出版社,2007.

[12] ERICKSON J. 黑客之道:漏洞发掘的艺术[M]. 2 版. 范书义,田玉敏,等译. 北京:中国水利水电出版社,2009.

[13] 王清,等. 0day 安全:软件漏洞分析技术[M]. 2 版. 北京:电子工业出版社,2011.

[14] HUSTON S D, JOHNSON J CE, SYYID U. ACE 程序员指南:网络与系统编程的实用设计模式[M]. 马维达,译. 北京:中国电力出版社,2004.

[15] 潘荣. ACE 技术内幕:深入解析 ACE 架构设计与实现原理[M]. 北京:机械工业出版社,2012.

[16] 杨奎武,郑康锋,张冬梅,等. 物联网安全理论与技术[M]. 北京:电子工业出版社,2017.

[17] 鞠卫国,张云帆,乔爱锋,等. SDN/NFV:重构网络架构 建设未来网络[M]. 北京:人民邮电出版社,2017.

[18] 朱河清,梁存铭,胡雪焜,等. 深入浅出 DPDK[M]. 北京:机械工业出版社,2016.

[19] 张瑜. Rootkit 隐遁攻击技术及其防范[M]. 北京:电子工业出版社,2017.
[20] 叶毓睿,雷迎春,李炫辉,等. 软件定义存储:原理、实践与生态[M]. 北京:机械工业出版社,2016.
[21] 闫长江,吴东君,熊怡. SDN 原理解析:转控分离的 SDN 架构[M]. 北京:人民邮电出版社,2016.
[22] 晃通,宫永直树,岩田淳. 图解 OpenFlow[M]. 李战军,薛文玲,译. 北京:人民邮电出版社,2016.
[23] 张焕国,赵波,等. 可信计算[M]. 武汉:武汉大学出版社,2011.
[24] 徐永士,王新华. 大数据时代下的通信需求:TCP 传输原理与优化[M]. 北京:电子工业出版社,2015.